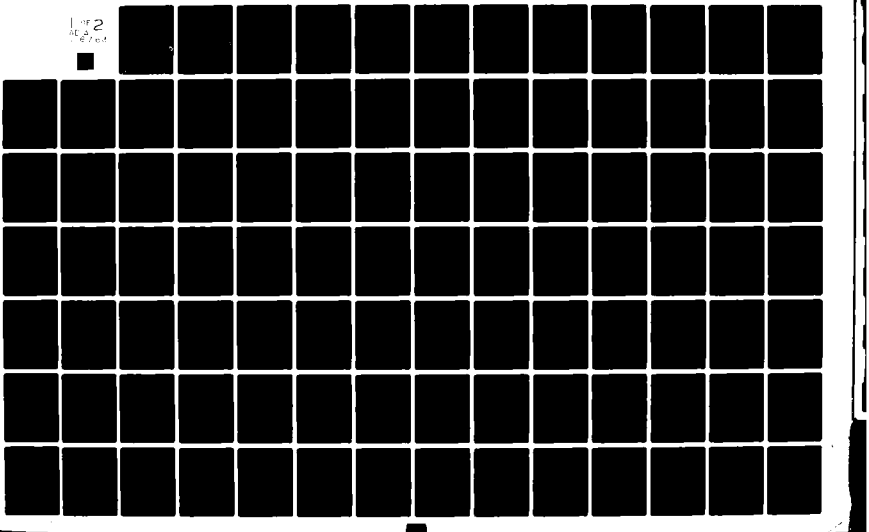
AD-A116 763 CONNECTICUT UNIV STORRS LAB FOR COMPUTER SCIENCE RE--ETC F/G 9/2
A DISTRIBUTED OPERATING SYSTEM FOR BMD APPLICATIONS.(U)
1982 B GAJIC DASG60-79-C-0117
UNCLASSIFIED TR-CS-82-4 NL



AD A116763

2

COMPUTER SCIENCE TECHNICAL REPORT

Laboratory for Computer Science Research
The University of Connecticut

DTIC
ELECTE
JUL 9 1982
D
H

COMPUTER SCIENCE DIVISION

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Electrical Engineering and Computer Science Department
U-157
The University of Connecticut
Storrs, Connecticut 06268

DTIC FILE COPY

82 06 18 025

(2)

A Distributed Operating System

For BMD Applications

Branimir Gajic

Technical Report CS-82-4

DASG 60-82-C-0117

DTIC
ELECTE
JUL 9 1982
D
H

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

A DISTRIBUTED OPERATING SYSTEM FOR BMD APPLICATIONS

Branimir Gajic

B.S., University of Belgrade, Yugoslavia, 1977

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at

The University of Connecticut

1982

DTIC COPY INSPECTED 2

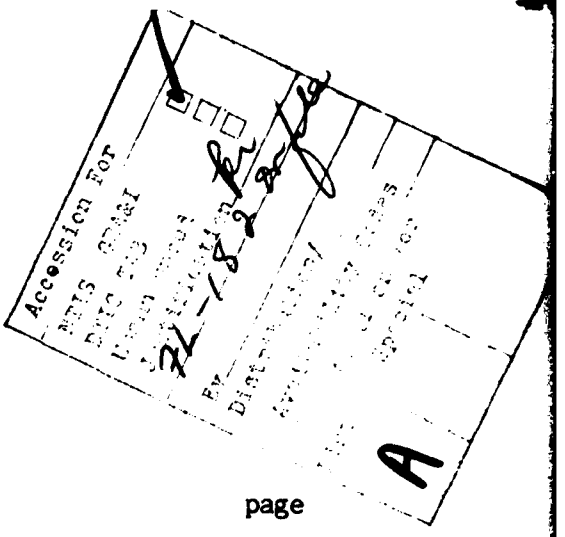


TABLE OF CONTENTS

LIST OF FIGURES

Chapter I

INTRODUCTION

The main objective of this research was to design and evaluate decentralized operating system concepts that would support real time BMD (Ballistic Missile Defense) application software executing on a distributed computing system with local and shared memories. BMD applications perform radar control and scheduling, target acquisition and processing, and intercept planning, in BMD tactical environment. These applications require exceptional level of performance, high reliability and continuous operation.

Logical distribution of the application software implies partitioning of the program into a number of small, cooperating units (processes [Brin78], [Hoar78], activities [Oust80], tasks [Bask77]). Process, as the unit of partitioning and distribution adopted here, is independent entity that closely cooperates with other processes in achomplishing a single task. There is no imposed 'parent - child' relationship; once a process is created it runs concurrently with other processes. Each process may freely, by its own will, engage in communication with others.

Close process cooperation is supported by the operating system kernel ([Font80],[Balk80]) which provides the virtual machine for application software. Message passing primitives allow for process communication and synchronization. There is no buffering of messages by kernel. The message passing construct uses a "randezvous" scheme: process that is ready to communicate first, is blocked until the other process is ready for communication.

Partitioned application software in the form of a number of processes is mapped to distributed hardware structure. As defined in [Ensl78], a distributed computer system is characterized by multiplicity of dynamically assignable general purpose resource components. This research results in specific hardware architecture which is proposed in Chapter III. The architecture consists of N nodes, each node containing processors, a module of local memory, and a module of global memory. The nodes of the distributed system are interconnected by a switching network of the "IDDR - Regular Network" type as defined in the taxonomy of computer interconnection structures [Ande75].

Integration of physical and logical components of the distributed system is done by distributed kernel and the set of proposed basic operating system functions. The primary issue is the control of low level hardware resources which include processors, local memories, global memory and the

switching network, as well as support of application software structures. Extensions to the Kernel, togather with two new operating system functions, namely the Memory Manager, and the File Manager, are proposed in Chapter IV. The two new functions perform allocation and deallocation of segments of global memory for data objects and file objects, and perform access to global memory for application processes. The Kernel, Memory Manager, File Manager, and the Network Interface, are togather referred to as the Basic Real-Time Operating System (the Basic RTOS).

The distributed programming language EPL [May 79] is extended with new language primitives. The syntax and semantics of new primitives is introduced in Chapter IV and appendices C, D, and E. The new EPL primitives allow application processes to invoke functions of the Basic RTOS, and to acquire/release logical and physical resources of the distributed system.

Performance of the BMD application software and performance of the operating system itself, were the most important issues in meeting the proposed objectives of this research. Therefore, the approach taken during the research was:

1. To explore particular structure of the BMD application software and to propose operating system functions that would take advantage of that structure.

2. To adapt the original operating system kernel functions to the shared memory architecture, and to propose new functions that would take advantage of the particular hardware structure.

Chapter II

BACKGROUND

Tnis chapter briefly overviews the basic concepts and underlying issues of real time environments. An effort is made to summarize from the current literature, the up to date experiences with development of the BMD applications (in order to learn about particular structures of BMD software), and supporting real time operating system. The structure of the sample BMD application is given in Appendix A, with description of its constituent subfunctions.

The communication network selected to interconnect physical and logical resources of the distributed architecture proposed in Chapter III, is the Cube Network [McMi80]. The formal definition of the network, description of its building blocks and modes of operation are presented in Appendix B. This chapter overviews the work done in that field and discusses some properties of that class of networks.

Extension of kernel functions and developement of new language primitives that will result in enhanced performance of the real-time application software, are the main focus of this investigation. At the end of this chapter, the properties of the distributed programming language EPL [May 79]

are summarized and the current state of the kernel [Balk80] is overviewd.

2.1 REAL-TIME ENVIRONMENT

2.1.1 Overview

A real time environment is characterized by processing activity triggered by randomly accepted external events [Lori81]. The processing activity for a particular event consists of a sequence of tasks, each of which must be accomplished within critical timing constraints. Computer system that operates in real-time environment is completely dedicated to the application. It is configured to guarantee response within timing constraints for worst case loads and is usually taylored for particular applications.

Real-time programs must be simple, reliable and highly efficient. From a language designer's point of view [Brin78] real-time programs have the following characteristics:

1. A real-time program interacts with an environment in which many things happen simultaneously.

2. A real-time program must respond to a variety of nondeterministic requests from its environment. The program can not predict the order in which these requests will be made, but must respond to them within certain time limits.

3. A real-time program controls a computer with fixed configuration and in most cases, performs a fixed number of concurrent tasks in its environment.

4. A real-time program never terminates.

Therefore, the language choosen for implementation of a real-time program must have capability to specify the fixed or variable number of concurrent processes, and must allow for nondeterminism in their interactions. In order to make programs understandable and manageable, processes must have simple control structures. Distributed computer systems may provide sufficient processing capability to permit simple program control structures in real time applications. In such a system the number of processes is equal or close to the number of processors.

2.1.2 BMD Application Software

The real-time systems for BMD applications have the general feed-back structure. The system accepts the status information about outside environment from sensors. It performs a sequence of computational tasks that will update system's perception of outside environment, and issue control signals that will adjust activity of sensors according to the updated perception. This structure suggests that the system can be viewd as a macropipeline of modules through which information flows unidirectionally.

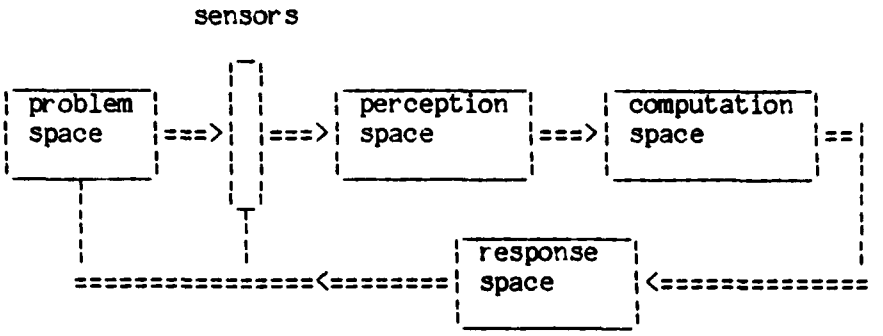


Figure 1: The System Space Components

BMD application systems are composed of three principal components [Vick79]: the sensor, the information processor, and the responsive device. The overall System Space can be partitioned into the following subsystems (fig. 1):

1. The Problem Space is the precise description of the environment in which the system operates. Problem Space is not the "picture" of the environment per se, but rather a set of qualitative and quantitative descriptions for objects of interest to the system.

2. The Perception Space is a set of sensor's perceptions of the Problem Space. The Perception Space describes objects of interest and dynamic structure of the environment (speed, height, distance, spatial relation of objects etc.)

3. The Computation Space transforms the elements of the Perception Space to the set of responses to sensors and to the Problem Space. Resources of the Computation Space are distributed hardware system, the set of processes of the application software, and the supporting real-time operating system.

4. The Response Space is the set of all responses provided by the Computation Space.

The focus of this investigation is on the Computation Space. The Computation Space can be viewed as a macro-pipeline of application processes (supported by the distributed real-time operating system and underlying hardware), the data sets that processes operate on, and a control scheme that activates processes and routes data sets. The structure of a sample Computation Space is presented in Appendix A.

Majority of the work reported in the area of BMD application systems deals with dynamic reconfiguration of distributed computing systems [Vick79], distributed real-time operating systems that support reconfiguration [Gree80], [Maju80], [Lee 80], application structures [SDCa81], and application requirements engineering methodology [Alfo77].

2.2 THE COMMUNICATION NETWORK

In a distributed system with local and shared memories the communication network is used to interconnect processing elements and memory modules. A processing element (PE) consists of a processor connected to a local memory. The switching network is used to support communication between processors and allow processors to acess memory modules of the global (shared) memory.

One way to interconnect M processing elements and memory modules is to use crossbar switch. It would allow simultaneous communication among processors or simultaneous access to different modules of shared memory. The problem is that the cost of the crossbar switch network grows with M^2, which makes it too expencive for distributed systems with large number of processing elements and memory modules.

A global bus architecture assumes a number of PE's and shared memories connected by a common bus. Acess to the bus is shared among processors by some allocation scheme (e.g.: Carrier Sense Multiple Acess with Collision Detection - CSMA/CD, or Time Division Multiplexing - TDM). Messages are sent from the source site onto the bus to be accepted by the destination site. The need for frequent communication among processing elements and simultaneous access to shared memory modules would result in considerable bus contension and push the demand on bus bandwidth.

The Generalized Cube Network [Mill80] is a multistage network that allows simultaneous communication as oposed to global bus architecture, but does not incurr the cost of crossbar switch network. The formalized description of the network is given in Appendix B. This network consists of N inputs, N outputs and $\log_2 N$ stages, each stage having N/2 interchange boxes. Each interchange box has two inputs and two outputs and routs incomming messages onto one of two alternative output paths, thus effecting an indirect communication among inputs and outputs of the network. Control of the netwotk is decentralized; each interchange box is independantly controlled by routing tags, which results in decentralized transfer control method. Message transfer paths between interchange boxes are dedicated. These characteristics make the Generalized Cube Network be classified as "IDDR - Regular Network" in the taxonomy of computer interconnection structures [Ande75].

The contention in the Generalized Cube Network is significantly less than contention in shared bus architectures, assuming the same load on the network. The reason for that is the fact that the cube network allows simultaneous input - output connections that are "conflict free". A conflict free, simultaneous connection of all inputs to all outputs is called the "passable permutation". For a network of size N (N inputs and N outputs) with two-function interchange boxes, the number of passable permutations is $2^{(N/2)\log_2 N}$

where $(N/2)\log_2 N$ is the number of interchange boxes in the networ [Chua80].

Discussion on different types of multistage networks and their parameters, can be found in [Sieg79] and [Muel79]. Implementation details and modes of operation of the Hybrid Cube Network, which exploits the advantages of two network configurations: processor to memory and PE to PE, are treated in [Mill80] and [Sieg80]. Algorithms for fault diagnosis in multistage networks, and hardware implementation of those algorithms, are described in [Rath80].

2.3 THE SYSTEM LEVELS

The main function of a distributed operating system kernel is to provide a virtual machine for application programs. The kernel acts as an interface between a high level programming language and distributed hardware, hiding the actual hardware configuration from the supported application software. The application software consists of a number of processes assigned to processing elements of distributed computer system. Each processing element must have a copy of the kernel in order to implement processes and support their interactions.

The kernel has no knowledge of communication network. The Communication Subsystem is the interface between the kernel and the communication network. It contains all network

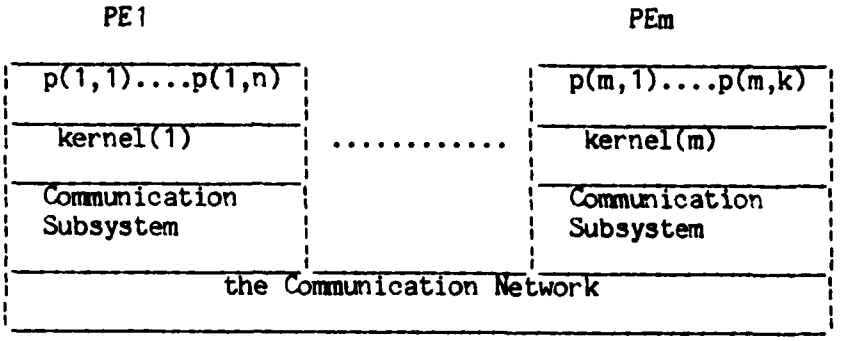


Figure 2: The System Levels

dependent software and performs the communication protocol ('link-level protocol') necessary to establish communication between processing sites. The system levels are illustrated on fig. 2.

2.3.1 EPL

EPL (Experimental Programming Language) [May 79] was choosen in the ongoing research as the high level programming language for writting distributed application software. EPL programs are expressed in terms of actors which are autonomous software processes. An actor is an instance of an act. Acts are reentrant segments of code so that the multiple instances of the same act can exist at the same time, sharing the same code. There is no compile-time upper bound on number of actors that can exist in the system. The run-time upper bound on the number of actors is defined by the available resources.

Actors interact by sending messages. The message passing construct uses blocking send and blocking receive so that messages serve both for communication between actors and synchronization of their activities. There is no buffering of messages by kernel. A message is directly coppied from the senders data area into receiver's data area. Synchronization is implied by the fact that the actor that is ready to communicate first is blocked until the other actor is ready for communication.

Actors may dynamically create other actors. There is no implied parential relationship between actors. Life span of an actor is from its creation until it reaches the end of code or terminates itself. It is possible to create any network of actors in a distributed system in order to provide paralelism and cooperation.

EPL is the language that manipulates values. Values are represented as bit patterns in one computer word. A computer word is the basic data object. EPL is typeless so that a word can be regarded as a bit pattern, a number, or the name of an actor. These characteristics give EPL the versatility and power for use in experimentation, but they do not provide for the protection of users.

The EPL language definition can be found in [May 79]. Extensions to the programming language, that provide support for fault tolerance, are summarized in [Balk80]. EPLUS

[Souz80] is a modification of EPL that is easier to read and understand by humans. The EPLUS compiler is more efficient and takes less memory space than EPL compiler.

2.3.2 The Kernel

The kernel is that part of an operating system that provides the run time environment for application processes. Each application process is a finite state machine, each state being the unique qualifier of a distinctive execution phase of the process. During its lifetime, a process will make transitions among different states, ie. running (executing on CPU), waiting to receive a message from another process, sending a message to another process, creating a new process, etc. The set of states that a process visits during its lifetime is a subset of a set of all states maintained by the kernel. The set of a process's states and relative frequency of state transitions are defined by the process's code segment.

The kernel must define an exhaustive set of process states in order to be able to distinguish between different execution phases of a process. It also has to maintain the consistent state information for each process in order to support interactions between processes. For a uniprocessor system, the kernel maintains the state information for overall system. In a distributed system each processing element has a copy of the kernel. Here, the system state infotmation

may be replicated on each processing element, or partitioned (distributed) among diferent processing elements.

Replication of the system state information results in considerable overhead due to the need to maintain the consistent state of the system. Partitioning of the system state information implies distributed knowledge of the system state. In the later case each kernel knows only about processes that are resident on the same processing site. Therefore, it must communicate with kernels resident on other sites in order to get information about state of non-resident processes.

The kernel consists of a set of functions that implement language primitives and provide for multiplexing of the CPU among processes. For general purpose operating systems, the policy of processor multiplexing should be separated from the implementation mechanism. In that case the processor multiplexing is implemented within kernel, whereas the policy of multiplexing is defined within scheduler. During this investigation the FIFO policy was assumed, so it was implemented within the kernel, which results in less operating system overhead. Extension from FIFO to priority scheduling can be done in a straightforward way.

The structure and organization of the kernel for distributed computer system with no common memory was investigated in [Font80]. The distributd system is based on multiple

processors directly connected by one or more communication links. The primary goal was the simplicity of kernel design. The results show considerably increased complexity of the multiprocessor version of the kernel, as compared to a single processor version. The greater complexity is reflected in increased number of instructions of kernel code needed to implement EPL primitives, and increased number of process states.

Modifications in the hardware structure of the distributed system resulted in the new version of the kernel. The resulting distributed architecture consists of multiple processors connected by a contention bus. The new kernel incorporates modifications that provide for fault tolerant EPL and explicit assignment of processes to CPU's [Balk80], with the "virtual channel" concept and the flow control algorithm for implementation of the communication mechanism [Cohe81].

Chapter III

THE COMPUTATION SPACE

The Computation Space is a set of hardware and software resources that perform transformations of elements of the Perception Space into elements of the Response Space. The Computation Space consists of the distributed hardware architecture, the real-time operating system functions, and the applications software. The application software is viewed as a macropipeline of processes. The topology of the macropipeline reflects the structure of the application software, and is referred to as the Computation Graph.

This chapter proposes particular hardware structure for the distributed system, and analyzes the elements of the application macropipeline Computation Graph.

Important considerations with respect to the Computation Space are performance, capability for reconfiguration, and reliability. The system performance determines how fast the system can respond to nondeterministic events of the Problem Space. Measures of performance are port-to-port time and throughput. Port-to-port time is the time elapsed from the moment the Computation Space samples an element of the Perception Space, until the moment the response to the corre-

sponding event of the Problem Space is ready for transmission. The maximum allowable port-to-port time is bounded by the time from the moment an event in the Problem Space is interpreted by sensors, until it becomes critical factor in the Response Space [Vick79]. The system throughput determines the number of responses (radar schedules) per unit time.

Dynamic reconfiguration implies the ability of the Computation Space to change its structure in time as a response to changed loading conditions, or in presence of hardware/software errors. Changing loading conditions result in need for load balancing or optimization of resources over time in order to meet performance specifications. Presence of hardware or software errors must be met by the capability of the Computation Space to detect and isolate errors and to restructure itself as to be able to continue operation with degraded capacity (and degraded capability of meeting performance specifications). The third aspect of reconfiguration of the Computation Space are changes in the structure of the Computation Graph over longer periods of time as a result of modifications of application software.

The design of the real-time operating system functions was based on meeting some of considerations mentioned above. The system performance and the performance of the application software were the ultimate goal of the design. Changes

within the Kernel allow for dynamic reconfiguration of the application software for the purpose of load balancing. Protection mechanisms are included to allow application processes to maintain accurate state information of the Problem Space, and to fascilitate error detection and recovery of the application software.

3.1 ELEMENTS OF THE COMPUTATION SPACE

3.1.1 Hardware

The distributed hardware architecture proposed here, consists of N homogeneous processing elements and N modules of global memory, interconnected by unidirectional Cube Network as shown on fig. 3.

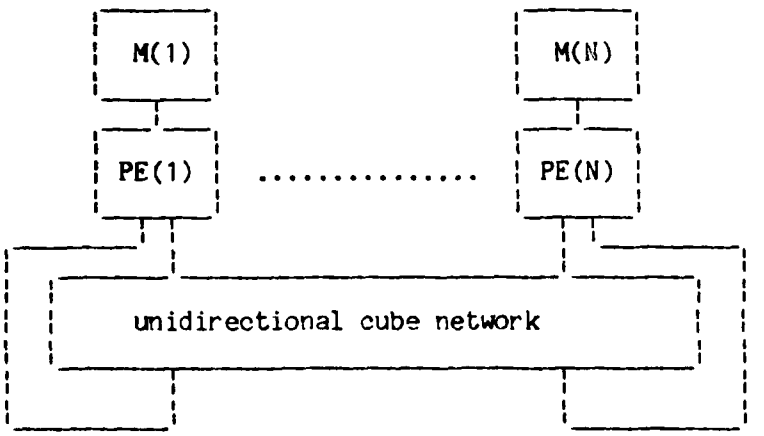


Figure 3: Topology of the Distributed System

Each of N modules of global memory is assigned to one of N processing elements; an access to global memory can be done only by a processor of the local processing element. Memory requests to nonlocal modules of global memory, result in exchange of messages between requesting PE and local PE. This is entirely general architecture, as opposed to systems with special purpose topologies determined by the application.

3.1.1.1 The Processing Element Configuration

The block diagram of a processing element, including the associated module of global memory, is shown on fig. 4.

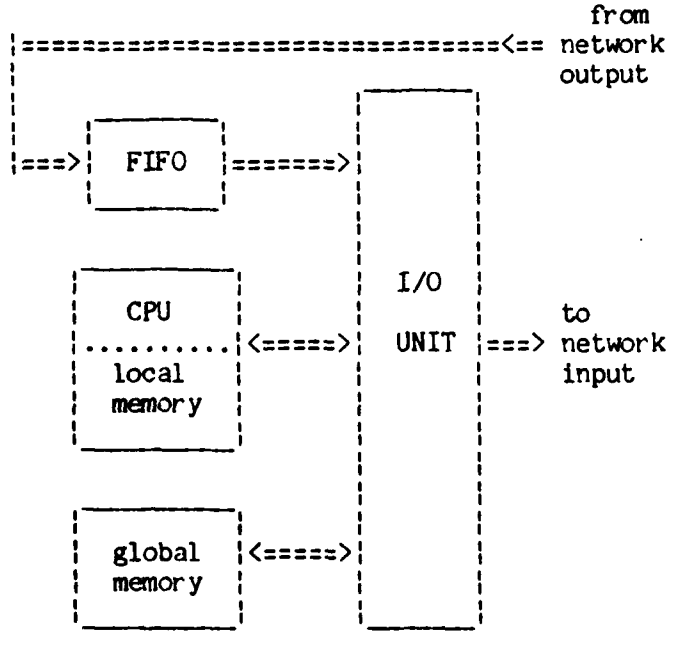


Figure 4: The Node of the Distributed System

This block diagram was suggested in [SDCc81], and consists of the processor with the local memory, module of global memory, Input/Output unit, and FIFO buffer. The CPU has no access to global memory. The function of the I/O unit is to service FIFO buffer, interpret and execute memory requests for the local module of global memory, and to serve as interface between the CPU and the communication network.

The nodes of the distributed system are to be built-up using elements of the Z8000 microprocessor family. The design considerations involved in selection of elements of the structure are:

1. to select technology that is currently available on the market,

2. to restrict access to local and global memories for purpose of protection,

3. to associate more inteligence with I/O unit for the purpose of performance and protection of data objects in global memory.

With this organization of the distributed hardware, the memory in the system can be divided into three levels, according to how fast the memory access can be performed. At the first level is the local memory which can be accessed by the CPU in 350-450 nsec. On the second level is the local module of the global memory, which can be accessed by the

CPU only through the I/O unit with incurred overhead. At the third level are nonlocal modules of global memory; access to any of them incurrs the overhead of three I/O unit processing, plus the network delay.

The first design consideration, to select only those elements of the Z8000 family that are currently available, constraints the mapping mechanism that translate logical adresses into physical adresses. In CM* ([Oust80]) the mapping is done by dedicated hardware that is more expensive than LSI-11 processor itself. This dedicated hardware (Slocal) is placed in processor's address path, which allows fast address mapping to nonlocal memory within the same cluster, but slows down access to local memory. K-maps are global switches dedicated to memory address mapping between different clusters. In CM* the access to memory in different cluster is 10 times as expencive as access to local memory, while the access to nonlocal memory within the same cluster is three times as expencive as access to local memory. Such fast mapping mechanisms are not available with the Z8000 family, which influenced the following decisions with respect to the management of the global memory:

1. application processes should be encouraged to read/write blocks of data; this decreases relative overhead of memory access.

2. The operating system should provide primitive functions that manipulate data/file segments in global memory for calling processes (like utility calls in general purpose operating systems); this considerably reduces overhead of access to global memory, as it reduces the traffic in the communication network.

The configuration of a node of the distributed system, as proposed here, is shown in fig. 5. The I/O unit is replaced with more powerfull Front-end Processing Unit (FPU), which is processor identical to the CPU (segmented Z8001).

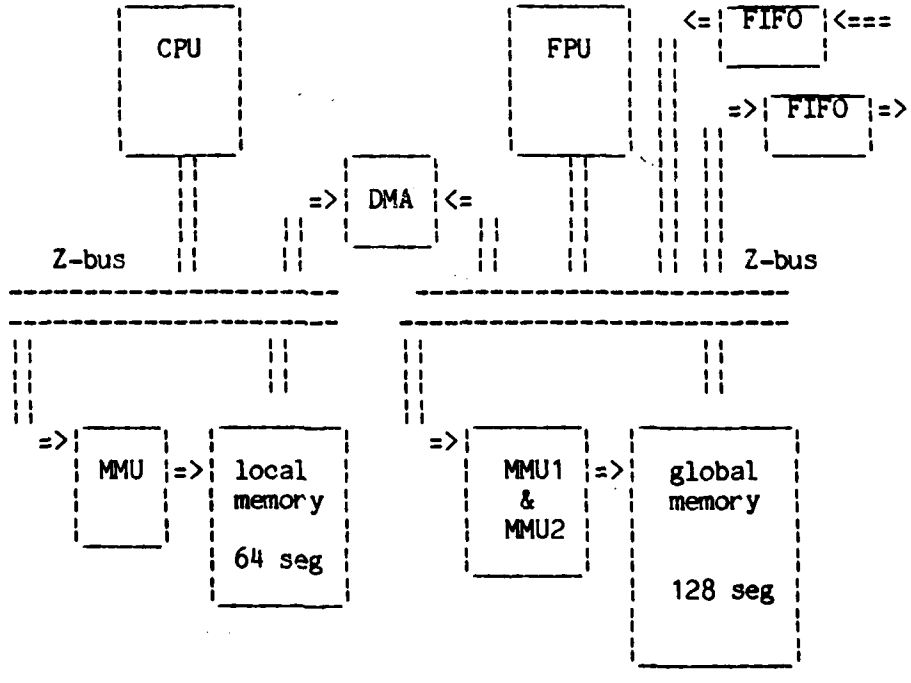


Figure 5: The Processing Element Organization

For reasons of protection, the CPU has no access to the coresident module of global memory, and FPU has no access to the local memory. The CPU and the FPU communicate by DMA link which consists of two two-channell DMA controllers (Z8016) working in cycle stealing mode. DMA controllers are interfaced to Z-bus on CPU side, and the Z-bus on the FPU side. The CPU initiates transfer on both channels of the DMA link [Soce80].

The CPU generates addresses that consist of a 7-bit segment number and 16-bit offset. Six bits of the segment number are used to select one of 64 Segment Descriptor Registers of the Memory Management Unit (Z8010 MMU) associated with the local memory. Each Segment Descriptor Register defines the base, the limit, and attributes of contiguous data segment in the local memory. The offset generated by the CPU is interpreted by the MMU as offset within the segment specified by the segment number [Zilo80].

Associated with the global memory module, are two MMUs that allow the FPU to specify up to 128 different segments in global memory; a 24-bits logical address space is mapped by MMUs into up to 8 Mbytes of physical memory. There would hardly ever be such a large storage requirement, but the fact that MMUs provide for hardware protection of data segments, led to the decision to seek large number of Segment Descriptor Registers (rather than excessively large

memory). The vast majority of data segments in global memory will be much smaller than the limit specified by corresponding segment descriptors.

Two FIFO buffers (Z8038 FIO) are interfaced to the front-end Z-bus and serve as message buffers between the node and the Cube Network. The input message buffer is connected to the last stage of the network, and the output buffer is connected to the first stage of the network.

3.1.1.2 The Communication Network

For the distributed system with N nodes, each node consists of one processing element and one module of global memory. The communication network is unidirectional, packet-switched Cube Network with $(N/2)\log_2 N$ two-function interchange boxes and $\log_2 N$ stages. Each interchange box is controlled independently through use of routing tags. For this distributed system, it may seem more advantageous to use one unidirectional Cube Network working in packet-switched mode, to support communication among processing elements, and the second, bidirectional Cube Network working in circuit-switched mode to support access to global memory (the Hybrid Cube Network, [McMi80]). Circuit switching is more efficient than packet switching when large blocks of data are to be transferred between global memory and a processing element. Once a path through the network is established, memory access is virtually as fast as access to local memory.

Implementation of such a network is more straightforward than implementation of packet switched networks, due to the fact that there is no need for complex network to memory interface. On the other hand, the network path width gets larger, considering that majority of processor's bus lines have to propagate through the network in order to access memory. With longer the bus lines, increasing mutual capacitance of lines results in slower propagation of signals and longer set-up times. The final result is somewhat slower access to the global memory than to the local memory.

In the case of circuit switching, the path through the network is established by passing the routing tag from the first stage of the network (stage $m-1$), to the last stage of the network (stage 0). As an interchange box in stage i receives the routing tag, it examines the tag and decides which interchange box in the stage i-1 should receive it, sends the tag to that interchange box, and holds the established connection (straight or exchange) between its inputs and outputs until the request for connection is dropped. The request signal is generated by the processor (memory request signal), and propagates from the stage m-1 of the network to stage 0 to which the module of the global memory is connected. Interchange boxes use request/grant handshaking signals to establish connections through the network.

Circuit switching considerably increases contention in the network due to the fact that complete path has to be established prior to data transfer between global memory and a processing element. As the number of passable permutations of a permutation network is much smaller than the number of all possible permutations (as application processes may randomly access any module of global memory), the number of requests for conflicting connections is expected to be high. The contention becomes worse with the fact that once established, a connection may be held for a relatively long period of time (as large blocks of data are transferred to or from the memory).

Another serious disadvantage of circuit switching is the posibility of deadlock. Network paths are resources of the distributed system; a process that deadlocks after 'tieing-up' a number of nodes of the network, can cause the total blocking of the network. Deadlock detection, prevention, or avoidance algorithms, would considerably degrade the performance of the real-time application software.

For reasons mentioned above, the packet switching mode of operation of the permutation network is seen as more suitable for the real-time application. It greatly reduces contention in the network, and is more efficient in supporting the communication among processes. It is less efficient for transfers of large ammounts of data to or from the global

memory, but it does not incurr the significant overhead of deadlock avoidance routines.

Selection of the packet switched mode of operation for access to global memory influenced the decision to have only one unidirectional, packet switched network, that will support communication among the nodes of the distributed system, as well as transfers of data between nodes and global memory. The FPU of each node serves as interface to global memory and interprets (and executes) each request for access to it.

It is assumed that there is a single network clock, connected to each interchange box in the network. In any given moment, an interchange box may be in request cycle, grant cycle, or transfer cycle. The request cycle consists of a set of events defined by values of logical levels on a number of control lines asserted by an interchange box in stage i, as it requests connection with an interchange box in stage i-1. The grant cycle consists of a set of events defined by logical levels on a number of control lines asserted by an interchange box in stage i, as it grants connection to an interchange box in stage i+1. The transfer cycle consists of one or more clock cycles, needed to transfer one word between two neighbouring stages of the network. For simplicity, it is assumed here that the network path width and the processor word are of the same size. Extension

to the case where the network path width is smaller than the processor word size, is simple and straightforward.

The network operates in asynchronous mode, so that in any given moment, an interchange box may be in any of the cycles mentioned above. Asynchronous mode of operation is more efficient than the synchronous mode (in synchronous mode of operation all interchange boxes are in the same cycle, simultaneously), as the total number of clock cycles needed for message transfer is smaller [McMi80].

A message consists of the routing tag, word-count that specifies the number of words to be transferred, and the content of the message. Each node saves the routing tag until complete message is transferred. The routing tag is a sequence of m bits, where m is the number of stages in the network. Each bit specifies what function the interchange box of the corresponding stage is to perform (straight - 0, or exchange - 1). The routing tag is calculated by the processor of the originating node, by XOR-ing the name of the source node and the name of the destination node [HoSi80].

The experience with MEDUSA, and results of simulation using the SIMNET model [SDCb81], show that degradation of system performance due to congestion in the network, may result with very frequent access to the global memory. On the other hand, as the relative frequency of access to global memory decreases, the contention in the network

decreases rapidly. Experience with MEDUSA shows that in order to maintain acceptible levels of performance, "the local-hit ratio" (the number of accesses to local memory for each access to global memory) must be better than 10:1. It clearly suggessts that process's code segment must not be executed from global memory.

Direct consequence of these results on the distributed system is that:

1. a process's local name space (code segment and data segment) must be resident in the local memory of the node executing the process.

2. when referencing global memory, a process should transfer blocks of data rather than individual words (it decreases traffic in the network).

It was stated in chapter II that the Cube Network (or any other permutation network) can be classified as "IDDR - Regular Network" according to taxonomy of computer interconnection structures proposed in [Ande75]. This classification is done considering the transfer path structure as dedicated, or accessible from only two points. Here, a path is defined as unidirectional point-to-point connection between two switching elements of the network. However, from a broader point of view, one may consider a path to be unidirectional connection between two nodes of the distributed system. In

the later case, the transfer path structure would change from dedicated to shared, which would cause the Cube Network to be classified as "IDS - Bus Window". This 'scope-of-view' dependancy appears as an unfortunate shortcoming of the proposed taxonomy.

There are some other characteristics of the Cube Network that help it be clearly recognized as Regular Network. One is very poor incremental growth where the size of the increment is N. The second characteristic is very low capability for reconfiguration in the case of the failure of an element of the network. The reconfiguration can be done only by having the whole sparing network, or changing to irregular structure which requires considerable software overhead.

3.1.2 The Application Software

The reason for designing the new set of operating system functions, and the corresponding set of new EPL primitives, was to take advantage of the particular structure of the application software in order to increase performance of the real-time system. The application software is structured as a macropipeline of processes through which information flows unidirectionaly. A particular structure of the macropipeline is referred to as the Computation Graph. Radar returns are correlated with corresponding radar schedules, and routed to one of four groups of processes ("processing threads") that perform search/verify processing, track initiation process-

ing, track state maintenance, or object classification processing and intercept planning.

The application processes maintain the state of the Problem Space within shared files (known object file, intercept plan file, radar schedule file etc.). Here, files are referred to as file objects. Information that flows through the macropipeline contains description of the state of elements of the Problem Space ('objects'), requests for radar schedules, enabling functions for other processes, etc. A unit of information that flows through the macropipeline is called the data object. (For example, information received by macropipeline after detection of n elements of the Problem Space during a single search, would consist of n data objects; each radar schedule is a single data object, etc.)

3.1.2.1 Process Name Space

A process's name space is a set of all file objects and data objects accessible by the process. It is implied that a processor executing the process, has access to its code segment. In HYDRA [Wulf74], when a procedure is invoked, a Local Name Space is created for that incarnation of the procedure. The Local Name Space is a list of capabilities that specify objects that running procedure can reference. A capability consists of a reference to an object, togeather with a collection of 'access rights' to that object.

Capabilities specify mapping from processes to objects, while backpointers specify mapping from objects to processes. Backpointers fascilitate exception handling; if an access to an object results in exception, all processes having access right to that object may be informed about the exception. If an object is moved in memory, backpointers are used to locate processes having capability for that object. Moving of objects in memory (for the purpose of memory management) results in additional overhead of maintaining capabilities. If capabilities are passes among processes with messages, much overhead is incurred in maintaining of backpointers.

Characteristics of the application software suggest that file and data object be static in global memory. File objects are created during System Generation, and are never destroyed. The file authorization table, effectively a backpointer list naming processes with right to access a file in read-write mode, is created with the file and associated with it. The process that creates a file (the 'owner process'), submits the list of processes allowed to access the file to the File Manager. The authorization list can not be changed afterwards.

A data object is static in global memory from its creation to its destruction. Names of data objects (pointers to objects) are passed with messages through the pipeline.

Processes can not posess names of data objects but only capabilities, and the Kernel performs mapping from capabilities to data objects. This protection mechanism secures that a process can not access a data object after it passed the capability for that data object to another process.

In MEDUSA [Oust80], each object is typed, each with defined set of its own type-specific operations. The total number of descriptors (capabilities) that may simultaneously exist for each object is specified when the object is created and can not exceed that number. In this real-time operating system data objects are not typed; each data object is treated as a sequence of uninterpreted memory words. On the other hand, file objects are typed, with unique type of 'file object' and a unique set of primitive operations implemented within the File Manager.

Data objects are created as an instance of information flow enters the macropipeline, and destroyed before it exits the macropipeline. The nature of the application software guarantees that there is no knowledge in the system about a data object once it is destroyed, as the process that destroyes the object must receive all capabilities for that object that exist in the system. This allows that data object names may be reused after objects are destroyed.

A process's name space (fig. 6.) consists of its code segment and local data segment that are resident in the

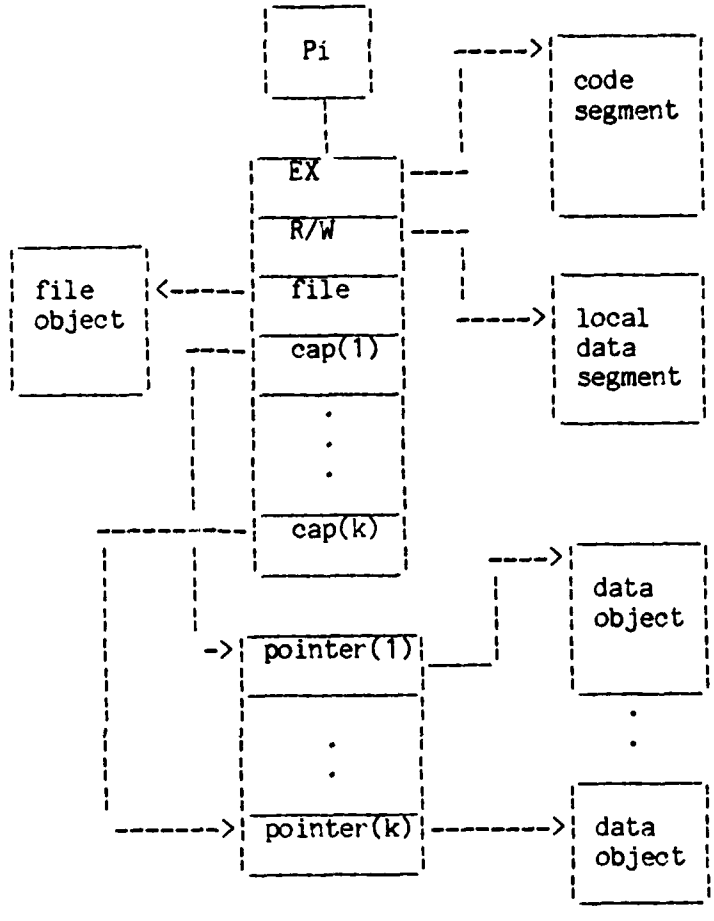


Figure 6: Process's Name Space

local memory of the node executing the process, and a number of data objects and file objects resident in global memory anywhere in the system. Access rights for access to the code segment ('execute') and the local data segment ('read-write') are imposed by the MMU of the local memory. Access rights to data objects are specified by the capability list which is mapped by the Kernel into the list of object names. There is a maximum number of capabilities a process may

hold, and it is the same for all processes in the system. Queuing of capabilities is done on EPL level (by application processes), while the queuing of corresponding pointers to data objects is done by the Kernel. Queuing reflects nondeterministic nature of the environment of the real-time system.

Access to a file is different from access to a data object in that any process may access any file object (if the process has the name of the file) in read-only mode. Only authorized processes may change the content of a file as they have read-write access to it. Access to files in the system is controlled and maintained by the File Manager.

3.1.2.2 Elements of the Computation Graph

The elements of the Computation Graph for an application macropipeline may be classified as:

Transient Node

For each instance of information flow sampled on its in-directed arc, the transient node of a linear macro-pipeline initiates one instance of information flow on its out-directed arc.

Absorption Node

For each n instances of information flow sampled on its in-directed arc, the absorbtion node of a linear

macropipeline initiates one instance of information flow on its out-directed arc.

Nonselective Fork

For each instance of information flow sampled on its in-directed arc, the node effecting Nonselective Fork within a nonlinear macropipeline, initiates one instance of information flow on each and every of its out-directed arcs.

Selective Fork

For each instance of information flow sampled on its in-directed arc, the node effecting Selective Fork within a nonlinear macropipeline, initiates one instance of information flow on each of a selective number of its out-directed arcs.

Nonselective Join

For each instance of information flow received on each and every of its in-directed arcs, the node effecting Nonselective Join within a nonlinear macropipeline, initiates one instance of information flow on its out-directed arc.

Selective Join

For each instance of information flow received on each

of a selective number of its in-directed arcs, the node effecting Selective Join within a nonlinear macropipeline, initiates one instance of information flow on its outdirected arc.

The 'instance of information flow' mentioned here, may consist of one or more capabilities, or one or more messages. It may differ from arc to arc within a macropipeline, and its entity is based on the consensus between the sending process and the receiving process. Each and every instance of information flow has a sequence number associated with it, which allows application processes to perform explicit ordering of events in the system (e.g. a process effecting Selective Join within the application's macropipeline Computation Graph).

3.1.3 The Real-Time Operating System

The Basic Real-Time Operating System (RTOS) consists of four functional subsystems: the Kernel, Memory Manager, File Manager, and the Network Interface.

The Kernel is replicated in local memory of each node, and its function is to provide runtime environment for application processes. The Kernel implements processes, performs multiplexing of the CPU, supports communication among processes, and directs system calls to other Basic RTOS functions. The Kernel makes immediate environment of appli-

cation processes, in the sense that every system call executed by an application process invokes the Kernel. If the system call corresponds to the memory management function, or the file management function of the Basic RTOS, the Kernel communicates with Memory Manager or File Manager for behalf of the calling process.

Memory Management function is replicated on each node of the distributed system. Management of local memory space is done on the CPU side of each node, and consists of memory allocation for dynammically created processes, and deallocation of memory space for terminated processes. Management of global memory, the primary issue here, is done on EPU side of each node, and consists of dynammic allocation/deallocation of memory space in the local module of global memory, for data objects being dynammically created and destroyed.

File Management function creates file objects for application processes during System Generation, and guards file objects from application processes during the run of the application software in tactical environment. File Management function is replicated on the front-end side of each node. (The term System Generation is used here to denote a sequence of steps, specified by the EPL code, that are necessary to build the macropipeline structure on the distributed hardware, and is different from the sequence of steps

necessary to 'bring-up' the operating system, which is usually refered to as the system initialization.)

The Network Interface function receives messages from the local Kernel and if messages are outbound, writes them into output FIFO buffer. It also reads messages from the input FIFO buffer and if they are directed to the local Kernel, sends them to the CPU side. If a message is from local or nonlocal Kernel, and invokes the resident memory management or file management function, the Network Interface executes the call to the corresponding entry of the Memory Manager or the File Manager. The Network Interface must interface to DMA handlers on the front-end side of the node where it resides.

The four functional subsystems of the Basic RTOS are shown in fig. 7.

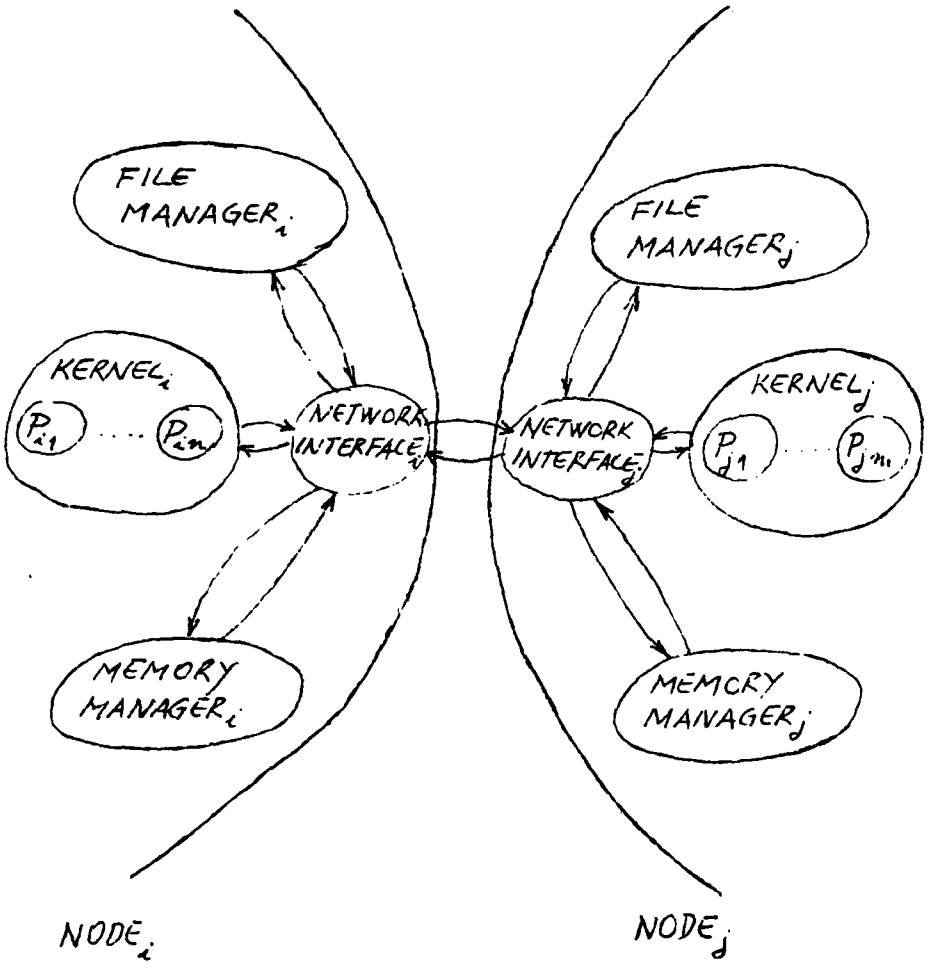


Figure 7: Functional Subsystems of the Basic RTOS

3.2 MEMORY LAYOUT

Resident in the local memory of each node of the distributed system are:

1. EPL code loaded at the same starting address of each node. Each EPL act has a unique, system wide name which is the starting address of its code.
2. Kernel code
3. Local memory manager
4. CPU side DMA handlers
5. Initialization routines

Resident in the global memory of each node of the distributed system are:

1. Memory manager
2. File manager
3. Network interface
4. FPU side DMA handlers
5. Initialization routines

Precise memory layout is hardware dependent and not important here. An example memory layout can be found in [Font81].

Chapter IV

THE OPERATING SYSTEM FUNCTIONS

This chapter overviews extensions to the EPL programing language that are intended to be used for writing BMD application software. The reason for introducing extensions to EPL are: to take advantage of the hardware architecture introduced in Chapter III, and to provide efficient support for elements of the Computation Graph of the BMD application software. Proposed extensions to the EPL result in two new operating system functions: memory management, and file management. The Kernel, Memory Manager, and the File Manager togather make the Basic Real-Time Operating System (referred to as the Basic RTOS). Extensions to EPL are described using the EPLUS syntax.

Extensions to EPL add new dimensions to interactions between application processes and the virtual machine defined by the Basic RTOS. A level of protection is added to the virtual machine interface; protection mechanisms are based on (and reach beyond) the fault detection mechanisms introduced in [Balk80], where it was assumed that all faults detected by the operating system Kernel, could be mapped into failures of virtual machine operations dealing with interactions among application processes. Protection mecha-

nisms, built into the Basic RTOS, imply increased cooperation among application processes and the operating system. In a possibly malicious environment, such as those supported by general purpose operating systems, the need for increased consensus between application software and the operating system, results in need for better protection in the virtual machine interface. In such systems, it is important to protect the operating system from user processes, and to protect user processes from each other. In the majority of systems it is done by imposing disjoint name spaces for all processes, and allowing access to shared objects only through the set of operating system calls (for example, access to pipes in UNIX [Ritc74]). Protection of user processes from operating system is rare, and of the second-order importance.

The reason for protection in the Basic RTOS is to achieve integrity of the whole system consisting of application processes, shared data objects, shared file objects, and the operating system functions. The reason for existance of protection mechanisms here, is not only to protect the system from ill behaved processes but also to allow for fault isolation and recovery of such processes. An application process whose virtual machine operation might undermine the integrity of the system, will find that virtual machine operation failed, and will be informed by the operating system about the reason for the failure. It is expected that

this feature prove usefull for error detection and recovery, especially during the development and testing of the BMD application software.

4.1 THE KERNEL

The Kernel of the Basic RTOS implements application processes and supports their interactions. Current Kernel calls (EPLUS primitives) are ([Souz80]):

```
PRODEC ptype [parms] [pbody]

PROCESS id <CPU expr> [exprlst] :: ptype <ONFAILURE stat>
 <, id <CPU expr> [exprlst] :: ptype <ONFAILURE stat>>*;

SEND expr [exprlst] <ONFAILURE stat>;

REPLY  [exprlst] <ONFAILURE stat>;

REC expr [exprlst] <ONFAILURE stat>;

RECF <expr> [exprlst] <ONFAILURE stat>;
```

A process is defined by PRODEC statement, where process body 'pbody' consists of a sequence of declarations, followed by a sequence of statements. Parameter list 'params' is a list of initialization parameters for the process, and the type of the process is assigned to 'ptype'. The PRODEC statement does not cause the process to begin execution.

Processes are created by the PROCESS statement. A process of type 'ptype' is created on the processor specified by the expression following the keyword CPU, or if the keyword is not specified, the process is created on the processor selected by the Kernel. The expression list 'exprlst' specifies initialization parameters to be received from the parent process. The PROCESS statement may be used to create more than one process at the same time.

Messages are exchanged among processes by message passing Kernel calls (SEND, REPLY, REC, RECF). Messages are values determined by evaluating expressions in a expression list 'exprlst'. A process executing SEND names the receiving process by the expression 'expr'; a process executing RECF names the sending process by the expression 'expr'. A process executing REC receives a message from any process that is ready to send to it; after the call, receiving process finds the message and the name of the sender on its data segment. A process executing REPLY call sends a message to the last process from which a message was received.

For all Kernel calls, if the keyword ONFAILURE is specified and the call results in error, the statement or the sequence of statements following ONFAILURE are executed. If the keyword ONFAILURE is not specified and the call results in error, the calling process terminates.

New Kernel calls fascilitate passing of capabilities along the Computation Graph of the application software. The process descriptor is extended to include the circular buffer containing names of data objects that the process has access to. A process can not own a data object name, but only capabilities which are pointers to the buffer with data object names. Access to the buffer is done only by the Kernel. This is used to prevent a process from having an access to a data object after it sent capability for that data object to another process. The extent of the buffer in the process descriptor is application dependent, and is equal to or greater than the maximum queue size of the application macropipeline. On the EPL level, application processes perform queuing of capabilities, while on the operating system level the Kernel performs queuing of corresponding data object names. The inherent FIFO characteristics of the application software allows for queuing done by the Kernel to be implemented in a straightforward way; it also allows the Kernel to monitor and impose FIFO policy on application processes.

The new process descriptor organization allows the Kernel to distinguish between passing of capabilities among processes, and passing of messages. The process descriptor organization, and the syntax and semantics of Kernel calls, are given in Appendix C. Here is the list of new EPL primitives that implement passing of capabilities among application processes:

SENDC expr [exprlst] <, expr [exprlst]>* <ONFAILURE stat>;

RECFC expr [exprlst] <, expr [exprlst]>* <ONFAILURE stat>;

RECC expr [exprlst] <, expr [exprlst]>* <ONFAILURE stat>;

SENDC sends capabilities specified by evaluating each expression list 'exprlst' to processes specified by evaluating each expression 'expr'. Sending process may send capabilities to up to 16 receiving processes, at the same time. RECFC receives capabilities from each of processes determined by evaluating expressions 'expr', and associates them with variables specified in expression list 'exprlst'. Receiving process may receive capabilities from up to 16 sending processes at the same time. RECC primitive receives capabilities from one of a set of processes specified by the receiving process by each expression 'expr'. Receiving process may specify up to 16 candidate senders at the same time, one of which will be selected as the sender.

Sending capabilities to a number of processes at the same time, and receiving capabilities from a number of processes at the same time, speeds up the information flow through the application macropipeline, and is expected to result in considerable increase of the performance of the application software. If a process is to send or receive capabilities from only one process at a time, its code segment would have

to specify a sequence of Kernel calls, instead of one SENDC or RECFC call as specified above. Such a sequence of Kernel calls results in deterministic ordering of process interactions; in this case a sending process would be blocked executing the first Kernel call in the sequence, if the corresponding receiver is not ready to receive, regardless of the fact that other receivers, named in subsequent Kernel calls, may already be blocked, waiting to receive from the sender. Deterministic ordering of process interactions increases the average time an application process is blocked, and consequently slows down the information flow through the macro-pipeline.

Syntax and semantics of SENDC and RECFC Kernel calls are intended to take advantage of nondeterministic nature of the BMD application software. More than one process may be specified as receiver or sender, and the order in which receivers (or senders) are specified in SENDC (or RECFC) does not imply the order in which capabilities are received (or sent). The Kernel corresident with the process executing SENDC, broadcasts enquiries to Kernels on all receiving sites, and sends data object names (that correspond to specified capabilities) as it receives positive acknowledgements. Semantics of RECFC and RECC supports nondeterminism in the same way.

In the case of sending capabilities to, or receiving capabilities from only one process at a time, the application process would have to specify a number of calls in a sequence, which would result in considerable overhead of context switching and Kernel's access to the list of ready processes. A process executing SENDC with six specified receivers, would save ten context switchings, as opposed to six sequential Kernel calls. Overhead saved by Kernel not having to access the list of ready processes, is expected to be much smaller, and is dependent on programming techniques.

The primitive SENDC performs the broadcast of capabilities, and is selected to implement fork structures of the application software Computation Graph. SENDC itself performs nonselective fork; the selective fork structure is easily implemented by enumerating all nonselective sub-forks within it.

The Kernel removes a data object name from the list of data objects that a process can access, after that process sends the corresponding capability to another process. The implementation of SENDC Kernel call must take into account the posibility that a process may broadcast the same capability to a number of processes. Therefore, capabilities should be 'taken away' from the source process only after all of them are sent to destination processes; otherwise,

errors with unforeseen consequences may ocurr (the Kernel may not find the capability, or it may send wrong capability to a destination process).

The primitive RECFC is used to implement nonselective join structure of the application software Computation Graph. As SENDC may broadcast capabilities for the same data object, RECFC must absorb capabilities that point to the same data object, in order to ensure that a process may have only one capability for a data object. That feature prevents a process from having access to a data object after it passed capability for that object to another process. Absorbtion of capabilities results in increased overhead within the Kernel as it maintains the list of data object names a process has access to. It is expected that the resulting overhead be much less significant than performance improvements gained by taking advantage of nondeterministic nature of the application software.

Selective join structure of the Computation Graph can be broken into a set of nonselective sub-join structures, and implemented using RECC primitive. Selective join reflects another aspect of nondeterminism of the application software: as information flows through the macropipeline, each node that routs (sends) information to its output arcs, makes the routing decision according to its interpretation of the information it received on its input arcs. On the

other hand, each node that receives information selectively on one or more arcs of the set of all input arcs, has no knowledge on which arcs it should receive that information (such apriori knowledge would negate the assumption of unidirectional flow of information through the macropipeline). These characteristics of the application software suggest that each process executing RECC primitive, thus effecting the selective join structure, should perform explicit ordering of instances of the information flow received on its input arcs, in order to maintain FIFO ordering of events within the system.

The Kernel supports dynamic environment in which processes may be created and may terminate. When a process terminates, its process descriptor and local data segment are reclaimed by the memory management function of the Kernel. The application software should be structured in such a way that a process terminating itself does not hold any capabilities, in order not to tie-up segments of global memory. Process names are never reused. A process name is 32-bit long; the eight least significant bits of the process name specify the node where the process resides. This allows up to 2^{24} different process names for each node, which ensures that process names will not be repeated during the run time of the application software.

Each Kernel in the distributed system has its own name, which is the eight-bit name of the node where the Kernel is resident. This knowledge allows Kernel to recognize coresident processes (checking the least significant byte of the process name) which eliminates overhead incurred by the current version of the Kernel when supporting interactions between coresident processes [Font81]. The current version of the Kernel passes enquiries, acknowledgements, and messages to the I/O Subsystem, even if the two interacting processes are resident on the same node.

The Kernel's knowledge of coresident processes may introduce bias in implementation of RECC primitive, in the sense that the implementation may favor coresident senders. This is the same bias that may appear in implementation of REC primitive, which may favor senders resident on lower-numbered nodes. This bias is avoided by introducing two pointers within the process descriptor. These two pointers point to nodes involved in the last execution of RECC or REC primitive, and allow for fair policy when selecting senders in the case of execution of the next RECC or REC primitive.

The table of coresident processes is maintained by the Kernel. That table maps 32-bit long process names into 16-bit pointers to process descriptors. The current organization of the MAP table [Cohe81] allows for fast, direct access to pointers to process descriptors. In the Basic

RTOS however, direct mapping from process names to process descriptors is not possible, and the MAP table should be organized as binary tree in order to minimize the search overhead within the Kernel. For a given number of entries in a table, the tree corresponding to binary search achieves the theoretical minimum number of comparisons that are necessary for search by means of key comparisons [Knut73]. As the number of processes on each node of the distributed system is small, binary search is expected to be more efficient than hashing with its overhead of computing hash function and collision resolution.

Error protection mechanisms are built into the Kernel to Kernel message protocols. Negative acknowledgement messages have associated Return Code which must be interpreted by the receiving Kernel. The Return code may specify that the application process is not ready for communication, or it may specify the type of error that was detected during the protocol. In the later case, the Kernel receiving negative acknowledgement, will pass the Return Code to the coresident application process involved in communication. The fact that the Kernel has to interpret Return Codes, introduces negligible overhead. The kernel does not interpret Return Codes resulting from File Manager and Memory Manager calls.

4.2 THE FILE SYSTEM

An important function of the Basic Real Time Operating System is file management. A file is an object which togather with other data and file objects define the name space of a process. In general purpose operating systems, the file manager maintains the file directory and provides information and access to files to the other functions of the operating system as well as user processes (UNIX, MEDUSA). The notion of a file in the Basic RTOS is similar to on-line files in MULTICS [Lori81] which can be referenced as objects without I/O interface. The file management function is replicated at the front end side of each node of the distributed system and has knowledge only about coresident files.

Files are permanent objects in global memory. They are created by application processes during the System Generation and are never destroyed or relocated. The name of a file is unique within the system and consists of the name of the node where the file is resident, the segment number and the file object identifier. The segment number uniquely associates one Segment Descriptor Register (of the pair of local Memory Management Units) with the file segment, thus specifying the base, extent, and attributes of the contiguous segment in the global memory.

The file system maintains the state information of the Problem Space. Each element of a file consists of a key, which is a unique identifier of the element within the file and possibly non-unique identifier of an element of the Problem Space, and the State Vector, which describes the current state of the element of the Problem Space. As each element of the Problem Space is identified by the State Vector of the same size, all elements of a file have the same length. Unlike UNIX or MEDUSA which view files as collections of uninterpreted bytes without imposing structure on them, the Basic RTOS maintains files as lists of structured elements.

The fact that a file is a collection of structured elements, raises the question of the consistency of the file organization as maintained by the file manager and accessed by application processes. The decision was to allow the access to a file only through the file manager, by the set of primitive operations. The implementation of primitive functions adds a new dimension of the protection mechanism. While the semantics of the key word ONFAILURE (extensions to EPL, [Balk80]) concentrates on fault detection in interprocess communication and process creation, here it is extended to include the protection mechanism.

The need for protection in the file system of the Basic RTOS is twofold. First, to maintain meaningfull and con-

sistent state information of the Problem Space, only authorized processes should be allowed to access a file. Therefore, the process that creates a file ("owner process"), must submit the list of processes authorized to access the file in read-write mode. The list of authorized processes, including the owner process, is associated with the file. Any process which can generate the file name may access the file in read-only mode. A read-write access to the file by an anauthorized process results in failure to complete the execution of the primitive, an the file manager returns to the calling process a code (Return Code) specifying the reason for the failure. This aspect of protection guards the file from contamination with invalid information, and thus prevents error propagation within the system.

The second aspect of protection detects inconsistent views of the file organization by processes accessing the file. Any process accessing a file is allowed to do so only on an element bases (for example, reading or writing only a portion of a State Vector is not allowed). The process must submit to the file manager arguments that identify that process's view of the file organization. If it differs from the real organization of the file, the call results in failure and Return Code specifies reason for the failure. This aspect of protection fascilitates the fast detection of errors within processes.

Extension of the fault detection mechanism in EPL to include protection greatly fascilitates error detection and recovery during the application software developement and testing phase. It is expected that these mechanisms may provide the basis for the error recovery during the run of the application software in the tactical environment, although the extensive error recovery may be prohibitive within the strict timing constraints of the real time application.

Following is the list and the brief description of proposed EPL primitives that make the file management function calls. The full semantics and message protocols are given in Appendix D.

```
ALLOCF id <exprlst> <ONFAILURE stat>;

DEL  expr <exprlst> <ONFAILURE stat>;

UPDATE expr <exprlst> <ONFAILURE stat>;

ADD  expr <exprlst> <ONFAILURE stat>;

COPY expr <exprlst> <onfailure stat>;
```

The primitive ALLOCF allocates contiguous segment in global memory of the same node where the calling process

resides, and associates a unique name of the segment with identifier 'id'. The list of arguments 'exprlst' specifies the extent of the file, identifies the file organization, and names processes authorized to access the file in the read-write mode.

The primitive DEL deletes an element from the file specified by 'expr'. The expression 'expr' must evaluate into a valid file name. The expression list 'exprlst' specifies the KEY of the element to be deleted. The file manager must search through the file for the specified KEY and if it is found, the corresponding element of the file is deleted. Only authorized processes may execute this primitive.

The primitive UPDATE updates the State Vector of the file element with the specified KEY. The expression 'expr' must evaluate into a valid file name. The expression list 'exprlst' specifies the KEY and the new State Vector. The file manager must search through the file for the specified KEY, and if it is found, the file manager will replace the old State Vector of the file element with the new State Vector. Only authorized processes may execute this primitive.

The primitive ADD adds an element to the file named with 'expr'. The expression list 'exprlst' specifies the State Vector and the size of the KEY of the new file element. The file manager adds the new element at the end of the file, and returns the KEY to the calling process. Only authorized processes may execute this primitive.

On the file management function call COPY, the file manager ccppies the specified KEY, State Vector, or entire element of the file named by the expression 'expr', and sends it to the calling process. Any process may execute this primitive, as long as it can generate the file name. The file element to be coppied is specified either explicitely, by naming its KEY or specifying that the last element of the file is to be coppied, or implicitely, by specifying the direction of the scan through the file. In the later case, each time the process invokes the file manager, the next item (the KEY, the State Vector, or both) in the sequence specified by the process (backward or forward),is copied. The forward sequense starts with the currently first element, and ends with the currently last element of the file. The backward sequence starts with the currently last element, and ends with the currently first element of the file. A process may terminate the scan before the last element in the sequence is reached.

At the beginning of the scan, as the response to the first call by a process for the forward/backward scan through the file, the file manager creates the File Access Control Table (FACT) for that process. The File Access Control Table specifies the name of the process, the direction of the scan through the file (forward or backward) and the current relative position of the process within the file. The relative position must be maintained as other processes

may simultaneously add new elements or delete old elements, while the specified process is advancing through the file. In the case of forward scan, each time an element before or including the relative position of the process is deleted, the relative position identifier is decremented by one. In the case of backward scan, each time an element is added at the end of the file, the relative position identifier is incremented by one, and each time an element after or including the relative position of the process is deleted, the relative position identifier is decremented by one. The memory manager increments the relative position identifier for the process during each COPY call. When the currently last item in a sequence is reached and coppied, the File Access Control Table for the calling process is destroyed, and the process is notified that the end of file is reached.

The File Access Control Table must be maintained by the file manager for each process that executes a sequence of COPY calls (thus scanning through the file) in order to maintain consistent sequence ordering in the presence of a number of other processes having the concurrent access to the same file. The maintenance of File Access Control Tables results in increased overhead within the file manager. The only way to avoid this overhead is to perform serialization of processes that have access to a shared file. Serialization would be done on EPL level, restricting processes to access common files only within Critical Regions (implemen-

tation of which is discussed in Chapter V). It is expected though, that serialization be much more disruptive to the flow through the application macropipeline than the overhead within the file manager.

One of the reasons behind the decision to maintain the File Access Control Tables is that searching through a file is done much more frequently than adding or deleting elements of that file. As files maintain the state information of the Problem Space, any possibly new element of the Problem Space must be first correlated to the existant state information, and any element to be dropped as redundant in identifying the state of the Problem Space, must first be correlated with the rest of the state information with respect to all aspects of the Problem Space. This fact suggests that the overhead incurred in maintaining the state of the File Access Control Tables, being proportional to the degree of concurrency of access to a common file by COPY-ing processes on one hand, and modifying processes on the other, is not considerable.

Serialization of processes is expected to have much more serious consequences to the flow of the application due to the fact that COPY-ing (reading) processes usually perform complex functions (for example, the Known Object Recognition function performs correlation of the State Vector of an unclassified element with State Vectors of all known ele-

ments of the Problem Space). To allow such processes exclusive access to a file, would lock out the file for prohibitively long periods, and prevent other processes from maintaining the accurate and updated state of the Problem Space. This would result in serious decrease in the performance of the application software.

Another reason to view the serialization as being much more disruptive to the flow of application than the overhead within the file manager, is the fact that larger the State Vector of a file element is, the relative overhead of maintaining the File Access Control Tables gets smaller, while the serialization results in much more serious performance degradation. All factors mentioned above, led to the decision to implement the file system such as to allow multiple concurrent "readers and writers". It must be noted however, that design considerations here differ from those in general purpose operating systems, where mutual exclusion of readers and writers guarantees file consistency. In the Basic Real Time Operating System, any access to a file is done through the file manager, which provides for low level serialization, file consistency, and protection.

4.3 MEMORY MANAGEMENT

Memory Manager is that part of the Basic RTOS which allocates segments of global memory for data objects, and performs read/write access to data objects for application

processes. A data object has a system-wide unique name. Data object name is 16-bit long; the low order byte specifies the node where the data object is resident, seven low order bits in the high order byte specify one of 128 Segment Descriptor Registers of the local pair of Memory Management Units, and the highest order bit identifies the data object as opposed to a file object. The Segment Descriptor Register identifies the base, extent, and attributes of the data segment.

A data object, as opposed to a file object, is unstructured sequence of memory words in global memory. Data objects encapsulate instances of information flow through the *application macropipeline*; pointers to data objects are passed along its Computation Graph. Application processes can not communicate using shared data objects, as opposed to MEDUSA or UNIX. Communication through shared data objects would incurr overhead of access to global memory and network propagation delay. The fact that access to global memory is expensive, was the main consideration beoynd the decision to use EPL message passing primitives for communication and synchronization of application processes. This eliminates the overhead of access to global memory, and may reduce the delay through the Cube Network.

Full description of the syntax and semantics of Memory Manager system calls, as well as Kernel to Memory Manager protocols, are presented in Appendix E. Here is their list and brief description of each of the calls:

```
ALLOC expr [exprlst] <ONFAILURE stat>;
FREE  expr <ONFAILURE stat>;
READ expr [exprlst] <ONFAILURE stat>;
WRITE expr [exprlst] <ONFAILURE stat>;
```

ALLOCF allocates contiguous segment of global memory for the calling process. Expression list 'exprlst' specifies the extent of the segment. Memory Manager returns the data object name to the Kernel which is coresident with the calling process. The Kernel saves the data object name in the process descriptor, and returns to the process the capability that points to the data object name. The capability is associated with the variable specified by 'expr'.

FREE destroyes the data object pointed to by the capability specified by the expression 'expr', and returns its space to the pool of free space in global memory.

READ reads a sequence of memory words from the specified data object, and sends them to the calling process. WRITE writes a number of conseccutive words for the calling process, into the specified object. Expression list 'exprlst' for READ and WRITE must specify offset within the data object, and the number of words to be read or written. Reading and writting blocks of words decreases relative overhead of access to global memory.

Protection mechanism within Memory Manager protects boundaries between objects. Any READ or WRITE that extends over the object boundary, results in segmentation exception message returned to the calling process. Segmentation exception message helps isolate ill-behaved processes, and fascilitates error detection and recovery of the application software. This is the only protection mechanism within Memory Manager, as application processes can freely read or write anywhere within a data object. It is possible to develop higher levels of protection by imposing types on data objects, but at the present time it is seen as being too restrictive for BMD application software.

4.4 SUMMARY

The set of new EPL primitives was introduced and implemented as to allow efficient interactions between application processes and the Basic RTOS. The Kernel makes the immediate environment of an application process, and it engages in communication with File Manager and Memory Manager in order to complete corresponding system calls for behalf of the calling process. New EPL primitives are implemented as to take advantage of the distributed hardware, and particular structure of the BMD application software.

New language primitives increased logical complexity of process management, and result in increased number of process states as compared to [Font80] and [Cohe81]. On the

other hand, new primitives allow for efficient utilization of system resources.

New aspects of protection and broader cooperation between application processes and operating system functions are introduced. Protection mechanisms allow the application software to maintain consistent state of the Problem Space. The state of the Problem Space is captured in shared data files, and shared data objects which encapsulate instances of information flow through application macropipeline. Increased cooperation between application processes and operating system allows for error detection and recovery.

Chapter V

TASK PARTITIONING AND PROCESS STRUCTURES

Application software may be structured as macropipeline of tasks of different complexities. For task allocation purposes it may prove advantageous to brake particular tasks into a set of subtasks as to allow multiplexing of complex tasks among a number of processing units, in order to exploit potential paralelism in the distributed system. On the other hand, partitioned task requires communication and synchronization among cooperating subtasks. Partitioning increases communication cost while increasing paralelism in computation, both of which will influence the performance of the application software. Wheather task partitioning will increase or decrease the performance of the application software, depends on the characteristics of the application program, task allocation policy, and the cost of implementation of the communication mechanism in the distributed system.

The Basic RTOS allows arbitrary structures of application processes. In MEDUSA, the task force is the fundamental unit of control; each task force is a collection of concurrent activities that closely cooperate in the execution of a single task [Oust80]. Activities are different from proc-

esses in that they can exist only as a part of a task force. EPL allows for writting distributed application software where the unit of partitioning and distribution is a process. EPL programs may specify arbitrary structures of processes; each process may be either independant entity, or a part of a group of processes cooperating on a single task.

Chapter IV introduced new EPL language primitives that allow application processes to invoke functions of the Basic RTOS. This chapter examines a way to use extended EPL to write and structure BMD application programs. The question of particular interest is how to use new Kernel calls to build elements of application's macropipeline Computation Graph. Examples of process structuring and task partitioning are given later in this chapter. Partitioning of complex BMD tasks into groups of subtasks is expected to fascilitate:

1. finer grain of resource sharing
2. finer grain of load balancing
3. robustness of BMD software in the presence of node failures
4. better performance of BMD programs

5.1 WORKER PROCESSES

Complex BMD application tasks may be partitioned into a number of worker processes. Administrator process [Gent80] hides worker processes; it receives work requests from other processes in the application macropipeline, and forwards those requests to available workers. Application processes have no knowledge about the structure of particular tasks consisting of administrator and workers; they know only administrator, and submit their work requests to it.

Additional worker processes increase the concurrency achieved in servicing work requests made to the administrator. The administrator maintains the list of available workers, or may delegate that duty to another worker process. The administrator does not perform SEND to a worker process; therefore it is never blocked waiting for available workers. It receives report from worker processes that a task is accomplished, and replies with a new task. The administrator is blocked only when there are no work requests from other application processes. In that case all worker processes are blocked as well, waiting for new work requests.

Because there are a finite number of workers, an administrator must either explicitly queue work requests when all workers are busy, or the aplication program may specify a queuing process that will queue work requests before it forwards them to the administrator. As suggested in [Gent80],

mutual exclusion follows because the administrator does not release workers unless their tasks are disjoint. Mutually exclusive worker processes allow concurrent servicing of different work requests. This is an unnecessary restriction in structuring of processes; later in this chapter it will be shown how worker processes may share data objects, using appropriate mechanisms to synchronize their activities.

The following segments of EPL code implement the queuing process, the administrator, and a number of worker processes. The queuing process receives work requests (capabilities) from processes a, b, or c, thus effecting selective join structure of the Computation Graph. The queuing process sends capabilities to the administrator, which forwards them to available worker processes. Administrator and the queuing process synchronize their activities by exchanging empty (null) capability.

```
PRODEC  queuing.proc [     ]
  [
   LET i=0;
   LET in=0;
   LET out=0;
   LET caller=0;
   VEC q max.dim;       /* the queue of work requests
                        /* (capabilities)

   /* receive the first work request and corresponding
   /* sequence number from either a, b, or c, and
   /* place them in the queue

   caller := RECC a [cap], b [cap], c [cap];
   RECF caller [seq.no];
   q!in := seq.no;
   q!((in+1)%max.dim) := cap;
   in := (in+2)%max.dim;
```

```
WHILE (true)
   [

/* selectively receive either work requests from a, b,
/* or c, or a signal from administrator to forward
/* the next work request

    caller := RECC a [cap], b [cap], c [cap], adm [ ];
    IF (caller /= adm)
        [

        /* the caller is either a, b, or c - queue
        /* the work request and the corresponding
        /* sequence number received from the caller

        RECF caller [seq.no];
        q!in := seq.no;
        q!((in+1)%max.dim) := cap;
        in := (in+2)%max.dim;

        IF (in=out)              /* block on full queue
            [
            RECFC adm [ ];
            SEND  adm [q!out];
            SENDC adm [q!(out+1)%max.dim];
            out := (out+2)%max.dim;
            ]
      ELSE
            [
            ]
        ]
    ELSE
       [

        /* the caller is administrator - forward the
        /* next work request and the sequence number

        SEND adm [q!out];
        SENDC adm [q!(out+1)%max.dim];
        out := (out+2)%max.dim;

        IF (in=out)              /* block on empty queue
           [
            caller := RECC a [cap], b [cap], c[cap];
            RECF caller [seq.no];
            q!in := seq.no;
            q!((in+1)%max.dim) := cap;
            in := (in+2)%max.dim;
            ]
        ]
   ]
]
```

The administrator maintains the list of available worker processes, and as soon as a worker process is available, it signals to the queuing process to forward another work request. It may be appropriate to mention here that the queuing process and the administrator are written in a way that maintains the 'need to know' principle. Specifically, administrator knows nothing about the queue of work requests, and the queuing process knows nothing about worker processes. As the queue of work requests becomes empty, the administrator will block itself trying to send the signal to the queuing process to forward next work request.

```
PRODEC adm [  ]
   [
   LET i=0;
   LET in=0;
   LET out=0;
   VEC w max.worker;  /* the list of worker processes

   FOR (i:=0; i<max.worker; i:=i+1)
      [

      /* create worker processes

      PROCESS serve CPU (i%max.cpu) [SELF] :: worker;
      w!i:=serve;
      ]

   SENDC queuing.proc [ ]; /* send signal to the
                           /* queuing process
   WHILE (true)
      [
      REC caller [seq.no];
      IF (caller = queuing.proc)
         [

         /* receive the next  work request from the
         /* queuing process, forward it to the next
         /* available  worker, and  update the list
         /* of  available  worker  processes
```

```
            RECFC caller [cap];
            SEND w!out [seq.no];
            SENDC w!out [cap];
            out := (out+1)%max.worker;
            IF (in=out)        /* no workers available
               [
               REC caller [ ];
               w!in := caller;
               in := (in+1)%max.worker;
               SENDC queuing.proc [ ];
               ]
            ELSE
               [
               SENDC queuing.proc [ ];
               ]
         ]
      ELSE
         [

         /* the  signal is received  from a worker
         /* process after it accomplished its task;
         /* place its name on the list of available
         /* workers

         w!in := caller;
         in := (in+1)%max.worker;
         ]
      ]
   ]
```

Each worker process writes its name in a data object pointed to by the capability it holds. A worker process writes its two word name at the offset specified by the content of the base word in the data object, and increments the base word by two. The code segment for worker processes does not reflect any particular characteristics of the application software; the intention here is to illustrate possible structuring of application processes.

```
PRODEC worker [adm]
   [
   LET m = 0;
```

```
LET i = 0;
LET j = 1;

WHILE (true)
    [
    RECF adm [seq.no];    /* receive the work
    RECFC adm [cap];      /* request from the
                          /* administrator

    m := READ cap [i,j]; /* read the base word
    WRITE cap [m, SELF]; /* write the workers
                          /* name at the offset
                          /* specified by the
                          /* base word
    m := m+2;
    WRITE cap [i,m];
    SEND sort [seq.no];  /* send the seq. number
    SENDC sort [cap];    /* and the capability
                          /* to the sorting process
    SEND adm [ ];        /* send the signal to the
                          /* administrator
    ]
]
```

As worker processes execute their tasks, they will block themselves trying to send signals to the administrator to forward new work requests. All tasks performed by worker processes are disjoint. After a worker process executes its task, it forwards the capability to the sorting process that maintains FIFO ordering of capabilities by sorting corresponding sequence numbers, and forwards them along the macropipeline. The code for the sorting process is not shown here.

The administrator may specify more than one type of worker processes. As it hides its workers, it hides the number of instances of each type of workers.

5.2 SYNCHRONIZATION

As task partitioning leads to structuring of the application software into groups of worker processes cooperating in execution of a single task, worker processes must be able to share data objects. The main issue here is synchronization of application processes sharing data objects. The synchronization task may be viewed as keeping the system in a 'legitimate state' [Dijk74], where each process step which could cause the transition of the system into an illegitimate state, must be preceded by a test deciding whether the process is to proceed or to be delayed. The result of the test depends on the content of system variables.

The variables that record the current state of a group of worker processes, may be resident in the common store (a portion of a shared data object), or distributed among those processes. In the first case, the worker processes must be granted mutually exclusive access to that portion of the shared data object that holds those variables. In the later case, the worker process wanting to make the step that might change the state of the system into an illegitimate state, must have permission from other worker processes in order to proceed (the permission is requested, and granted or denied, using the message passing primitives [Rica81]) or permission from the Kernel (by executing the corresponding Kernel primitives [Reed79]).

The following segment of EPL code implements mutual exclusion semaphore process, and somewhat modified version of the code for worker processes shown in the previous section. The mutual exclusion semaphore allows worker processes to serialize their access to shared data object. The base word of the shared data object is the 'state variable'; it specifies the offset within the data object where the process executing its critical region is to write its name. The critical region is the segment of the worker process code between two successive calls to the semaphore process. Worker processes will write their names in the shared data object, in the order in which they entered their critical regions.

```
PRODEC semaphore [adm]
   [
   WHILE (true)
     [
     REC caller [ ];
     RECF caller [ ];
     ]
   ]

PRODEC worker [adm]
   [
   LET m=0;
   LET i=0;
   LET j=1;

   WHILE (true)
     [
     RECFC adm [cap];
     SEND semaphore [ ]; /* enter the
                         /* critical region
     m := READ cap [i,j];
     WRITE cap [m,SELF];
     m : m+2
     WRITE cap [i,m];
```

```
        SEND semaphore [ ]; /* exit the
                            /* critical region
        SENDC sort [cap];
        SEND adm [ ];
        ]
    ]
```

It is important that the implementation of the REC primitive avoids bias in selecting the sender in the list of candidate senders (this matter is discussed in more detail in Chapter IV). The fair policy of selecting senders is important if the system is to provide worker processes with a fair mechanism to acquire system resources (in this particular case, to have exclusive access to shared data objects). It is important to note, however, that the implementation of REC primitive does not guarantee FIFO ordering of senders. Therefore, the semaphore process can guarantee the fair policy in selecting the next worker process to enter its critical region, but it can not guarantee FIFO ordering of those processes. The FIFO ordering of events in a distributed system may be maintained using message passing among competing processes ([Lamp78],[Rica81]).

An event is any change in the state of the system, like the change in an operating system variable, or a change in the state of one in a number of interacting processes. Synchronization of concurrent processes requires controlling the relative ordering of events in the processes [Reed79]. Processes can synchronize their activities by exchanging

messages or by using shared variables accessed within critical regions.

Recording and signaling of events in a distributed system may be implemented either by introducing the set of Kernel calls, or by using the communication primitives. The Basic RTOS allows application processes to synchronize their activities either by exchanging messages, or by using mutual exclusion to protect shared variables that control ordering of events. It does not provide system calls that would allow application processes to explicitely observe or signal events in the system. Whether these mechanisms are sufficient for BMD applications is a question to be answered by further investigation.

Chapter VI

CONCLUSION

The main goal of this research was to design and evaluate decentralized operating system concepts that will support real-time BMD applicaions executing on distributed hardware with local and shared memories. The objective was to develop real-time operating system functions that would perform efficient integration of distributed resources, and support execution of BMD application software with high levels of performance, reliability, and continuous operation. Results of current research efforts in the field of distributed hardware architecture, operating systems design, and distributed programming languages, are studied in order to identify major issues and evaluate proposed solutions.

This investigation resulted in a proposed configuration of the distributed system, the set of new operating system functions that together with an existing Kernel [Cohe81] make the Basic Real-Time Operating System, and the set of new EPL language primitives that provide BMD application processes with efficient mechanisms for communication, synchronization, and effective utilization of distributed system resources.

6.1 DISTRIBUTED HARDWARE

Distributed hardware consists of N nodes connected by unidirectional permutation network. Each node consists of a processing element (processor with local memory), and a module of global memory. Permutation network proposed is the Cube Network. The distributed hardware is organized so that the memory in the system is divided into three levels, according to how fast the memory access can be performed (local memory, the local module of global memory, and nonlocal module of global memory). Precise measurements of access times, and measurements of typical access rates for BMD application software, will convey important results about the performance of the distributed system. These measurements are foreseen as the most important step in the course of further investigation, immediately following the implementation of the system.

Different modes of operation of the Cube Network are considered and evaluated with respect to speed and simplicity of implementation. Results show that very frequent access to global memory would result in serious performance degradation of the application software. These results are used to propose a type of access to logical resources, and propose resource allocation policies, that would minimize the overhead of access to global memory.

6.2 OPERATING SYSTEM FUNCTIONS

The function of the Basic Real-Time Operating System is to integrate physical and logical resources of the distributed system, and to support BMD application software. The functions of the Basic RTOS are the Kernel, File Manager, Memory Manager, and the Network Interface.

The Kernel makes immediate environment of an application process. It implements processes and supports their interactions. It communicates with other Kernels in order to complete Kernel calls for behalf of the calling process. The Kernel communicates with File Manager and Memory Manager in order to complete File Manager calls and Memory Manager calls for the behalf of the calling process.

The File Manager guards file objects from user processes, and performs access to file objects for user processes. The Memory Manager manages the global memory of the distributed system. It allocates/deallocates segments of global memory for data objects, and performs access to data objects for application processes.

The Basic RTOS provides the virtual machine for BMD application processes. Two fundamental characteristics of the virtual machine interface are protection and increased cooperation between application processes on one hand, and the operating system functions on the other. Protection mechanisms guard logical resources of the distributed system

(shared file objects and shared data objects) and help isolate erroneous application processes. Increased cooperation in interactions between application software and the operating system helps convey information about exceptions to application processes, and fascilitates error recovery of the application software.

Protection mechanisms built into the File Management function of the Basic RTOS allow the BMD software to maintain consistent state information for the Problem Space, isolate erroneous processes, and attemt error recovery in the presence of such processes. Protection mechanisms built into the Memory Management function of the Basic RTOS, protect data object boundaries, and fascilitate error detection and recovery of the application software.

6.3 THE APPLICATION SOFTWARE

Characteristics of the BMD software are studied in order to take advantage of its particular structure. The application software is structured as a macropipeline of processes through which information flows unidirectionally. Units of information that flow along its Computation Graph are called data objects. Application processes maintain the state of the Problem Space in shared data files called file objects.

A process is the unit of partitioning and distribution of the BMD software. The notion of 'process name space' is

introduced to identify all file objects and data objects accessible by a process. Data objects are accessed through capabilities maintained by the Kernel. Capabilities are passed from a process to a process along the Computation Graph as information flows through the application macro-pipeline. Data and file objects are static in global memory. File objects are accessed directly, by their names.

Nondeterminism in interactions among processes of BMD application software, and particular structure of building elements of its Computation Graph, were investigated in order to propose efficient system calls that would take advantage of these characteristics.

6.4 THE NEW EPL PRIMITIVES

The distributed language EPL was extended to include new language primitives that invoke functions of the Basic RTOS. New primitives are introduced in a way that preserves the flavor of the EPL. The semantics of new primitives, however, are extended as to include new protection mechanisms that fascilitate error detection and recovery of the application software. Extensions to the semantics of new EPL primitives require increased cooperation between BMD software and the operating system.

6.5 FURTHER RESEARCH

The further work to be done in developing new operating system concepts for BMD applications can roughly be divided into three main areas: implementation, measurements, and programming techniques.

The implementation of concepts developed in this thesis is the first step in building the distributed system for BMD applications. Measurements performed on the operating system (like the speed of access to nonlocal memory, network delay, etc) should be used to validate (or invalidate) some of the assumptions here, and justify design decisions. Thorough measurements will give detailed performance characteristics of the operating system. Experimentation with programming of BMD software will evaluate design decisions conserning new language primitives, and measurements performed on BMD application software will yield important results on how sufficient new primitives are, and how efficiently they may be implemented.

REFERENCES

[Alfo77] Mack W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977.

[Ande75] George Anderson and Douglas Jensen, "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples", Computing Surveys, Vol. 7, No. 4, December 1975.

[Balk80] E. Balkovich, "Decentralized Systems", Technical Report CS-15-80, Labaratory for Computer Science Research, The University of Connecticut, 1980.

[Bask77] F. Baskett, J. Howard, and J. Montague, "Task Communication in DEMOS", Proceedings of Sixth ACM Symposium on Operating Systems Principles, November 1977.

[Boeb78] W. E. Boebert et al, "Kernel Primitives of the HXDP Executive", Proceedings of the IEEE Computer Cociety's 2nd International Computer Software and Applications Conference, November 1978.

[Brin78] Brinch Hansen, "Distributed Processes: a Concurrent Programming Concept", Comm. of the ACM, Vol. 21, No. 11, November 1978.

[Chua80] Chuan-lin Wu and Tse-yun Feng, "A Software Technique for Enhancing Performance of a Distributed Computing System", Proceedings of the IEEE Computer Cociety's 4th International Computer Software and Applications Conference, October 1980.

[Cohe81] Lawrence S. Cohen, "Communication Subsystem based on a CSMA/CD channel", MS Thesis, The University of Connecticut, December 1981.

[Dijk74] Edsger Dijkstra, "Self-stabilizing systems in Spite of Distributed Control", Comm. of the ACM, Vol. 17, No. 11, November 1974.

[Gent80] W. M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Admin-

istrator Concept", Software Practice and Experience, Vol. 11, No. 5, May 1981.

[Gree80] Michael L. Green et al., "A Distributed Real Time Operating System", Proceedings of the Symposium on Distributed Data Acquisition, Computing, and Control, Dec. 1980.

[Hoar74] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", Comm. of the ACM, Vol. 17, No. 10, October 1974.

[Hoar78] C.A.R. Hoare, "Communicating Sequential Processes", Comm. of the ACM, Vol. 21, No. 8, August 1978.

[HoSi80] H. J. Siegel and R. J. McMillen, "The Use of The Multistage Cube Network in a Multimicroprocessor Test Bed", School of El. Engineering Technical Report TR-EE 80-16, Purdue University, June 1980.

[Knut73] Donald E. Knuth, "The Art of Computer Programming", Vol. 3 / Sorting and Searching, Addison-Wesley 1973.

[Lamp78] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", Comm. of the ACM, Vol. 21, No. 7, July 1978.

[Lee 80] Edward Y. S. Lee et al., "A Distributed Data Base Manager", Proceedings of the Symposium on Distributed Data Acquisition, Computing, and Control, Dec. 1980.

[Lori81] H. Lorin and H. M. Deitel, "Operating Systems", Addison-Wesley, 1981.

[Madn74] Stuart E. Madnick and John J. Donovan, "Operating Systems", McGraw-Hill, 1974.

[Maju80] Samprakash Majumdar and Michael Green, "A Distributed Real Time Resource Manager", Proceedings of the Symposium on Distributed Data Acquisition, Computing, and Control, Dec. 1980.

[May 79] M.D. May and R.J.B. Taylor, "The EPL Programming Manual", Report No. 7, Department of Computer Science, University of Warwick, Coventry, England, 1979.

[McMi80] R. J. McMillen and H. J. Siegel, "The Hybrid Cube Network", Proceedings of the Symposium on Distributed Data Acquisition, Computing, and Control, Dec. 1980.

[Muel79] H. J. Siegel, R. J. McMillen and P. T. Mueller, "A Survey of Interconnection Methods for Reconfigurable Parallel Processing Systems", Proceedings of the 1979 National Computer Conference (NCC), June 1979.

[Oust80] John K. Ousterhout, "Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa", PhD Thesis, Carnegie-Mellon University, April 1980.

[Rath80] B. D. Rathi and M. Malek, "Fault Diagnosis of Interconnection Networks", Proceedings of the Symposium on Distributed Data Acquisition, Computing, and Control, Dec. 1980.

[Reed79] David Reed and Rajendra Kanodia, "Synchronization with Eventcounts and Sequencers", Comm. of the ACM, Vol. 22, No. 2, February 1979.

[Rica81] Glenn Ricart and Ashok Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks", Comm. of the ACM, Vol. 24, No. 1, January 1981.

[Ritc74] Dennis Ritchie and Ken Thompson, "The UNIX Time-Sharing System", Comm. of the ACM, Vol. 17, No. 7, July 1974.

[SDCa81] "An Example BMD Problem for Experimentation on Dynamically Reconfigurable Distributed Computing Systems", Technical Memo TM-HU-301/000/01, System Development Corporation, April 1981.

[SDCb81] Technical Memo TM-HU-303/001/00, System Development Corporation, July 1981.

[SDCc81] Technical Memo TM-HU-300/000/00, System Development Corporation, January 1981.

[Sieg79] Howard Jay Siegel, "Interconnection Networks for SIMD Machines", Computer, June 1979.

[Soce80] Alex Soceanu, "A Distributed Computing System", Class Project in Computer Architecture, The University of Connecticut, Spring 1980.

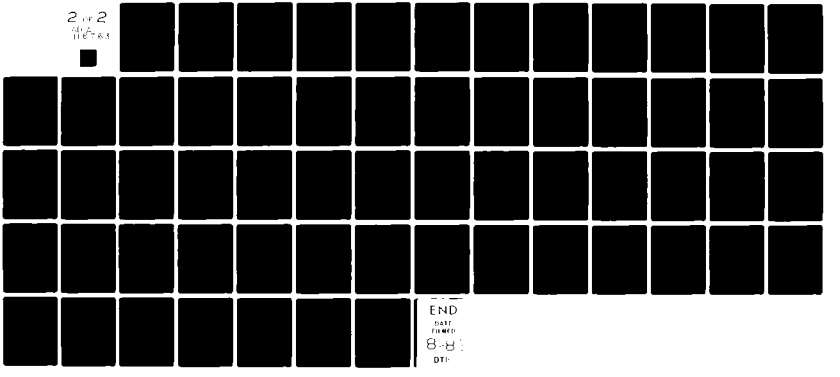
[Souz80] Robert Souza, EPLUS Language Manual, Labaratory for Computer Science Research, The University of Connecticut, 1980.

[Vick79] Charles R. Vick, "A Dynamically Reconfigurable Distributed Computing System", PhD Thesis, Auburn University, Auburn, Alabama, December 1979.

[Wulf74] W. Wulf et al., "HYDRA: The Kernel of a Multiprocessor Operating System", Comm. of the ACM, Vol 17, No. 6, June 1974.

[Zilo80] Zilog Microcomputer Components Data Book, February 1980.

AD-A116 763 CONNECTICUT UNIV STORRS LAB FOR COMPUTER SCIENCE RE--ETC F/6 9/2
A DISTRIBUTED OPERATING SYSTEM FOR BMD APPLICATIONS.(U)
1982 B GAJIC DASG60-79-C-0117
UNCLASSIFIED TR-CS-82-4 NL



Appendix A

A SAMPLE BMD APPLICATION STRUCTURE

The application structure shown in fig. 8. is a BMD problem decomposed into tasks that make up data processing threads required for real-time radar control, target acquisition and tracking, object classification and intercept planning. A full description of the system, with underlying assumptions, compiled program sizes, shared data base sizes and data item state vector sizes, is given in [SDCa81].

This application structure is the highest level of Computation Space as defined in chapter II. It must be supported by the real-time operating system and underlying distributed hardware. The overall BMD function is partitioned into the following tasks:

1. Radar Return Assimilation (RRA) receives radar returns, checks them for errors, correlates them with radar orders in the Radar Schedule File (RSF), and determines the next processing function based on pulse type.

2. Detection Returns Processing (DRP) stores search returns in Search Data File (SDF) and initiates

tracking request for each search return that is positively verified by verify pulse.

3. Track Initiate Sequence Request (TISR) creates a track initiate state vector for positively verified returns and requests track initiate pulse from the Radar Schedule Function (RTSA).

4. Track Initiation Processing (TIP) generates an initial target state estimate on TISR request. Target state estimate is improved after each Track Initiate (TI) return, requested byTrack Initiation Pulse Request (TIPR). When target's error statistics are small enough, the target is rejected by Cross Traffic Rejection (XTR) function or Known Object Recognition (KOR) function, or if not rejected it is considered a target track and its state maintenance (Track Processing) is requested from the Radar Schedule Function (RSF).

5. Object Track Processing (TKP) maintains the state information for target tracks through access to the Known Object File (KOF), and for a given object selectively invokes: Active Object Discrimination function (AODR) to request scheduling of the active discrimination pulse from the Radar Scheduler, Passive Object Discrimination function (POD), Impact Point Prediction function (IPP),

Track Prediction function (TPT) that will schedule track request from the radar scheduler, and Redundant Track Elimination function (RTE).

6. Active Object Discrimination (AOD) receives radar returns from active discrimination pulse and discriminate between target tracks and traffic decoys.

7. The Object Classification (CLS) task maintains Threatening Object File (TOF) and decides wheather to drop a track passing its identity to the Drop Track Processing (DTP) or to initiate intercept processing. In order to maintain consistent state information of the outside environment, DTP must have access to all system files that contain information about tracking objects.

8. Intercept Plan Generation (IPG) generates preliminary intercept plan for the most threatening objects when their tracking error statistics are sufficiently small.

9. Intercept Launch Request (ILR) transmits computed intercept point to the interceptor in the launch command.

10. Radar Template Slot Assignment (RTSA) or Radar Schedule function allocates a slot on the radar

timeline template to each pulse request, and according to the pulse type selectively forks Track Initiate Pulse Scheduler (TISK), Track Scheduler (TKSK) or Active Discrimination Scheduler (ADSK).

11. Radar Schedule Transmission (RPST) is enabled periodically by a signal from the radar to transmit the next frame of the radar schedule to the radar. A copy of the radar schedule is placed in the Radar Schedule File (RSF) to be used by RRA to correlate radar returns.

This application structure can be partitioned into four distinct functional groups, each of which contains one or more data processing threads. The functional groups are: search/verify processing, track initiation processing, track state maintenance and object classification processing, and intercept planning. The RRA function correlates radar returns with radar schedule commands and according to the required pulse type in radar schedule, selectively routes radar returns to one of the four functional groups.

At every point, the application system maintains the state of the monitored environment within its shared files:

1. SDF - Search Data File

2. KOF - Known Object File

3. TOF - Threatening Object File

4. IPF - Interceptor Plan File

5. DTF - Drop Track File

6. RSF - Radar Schedule File

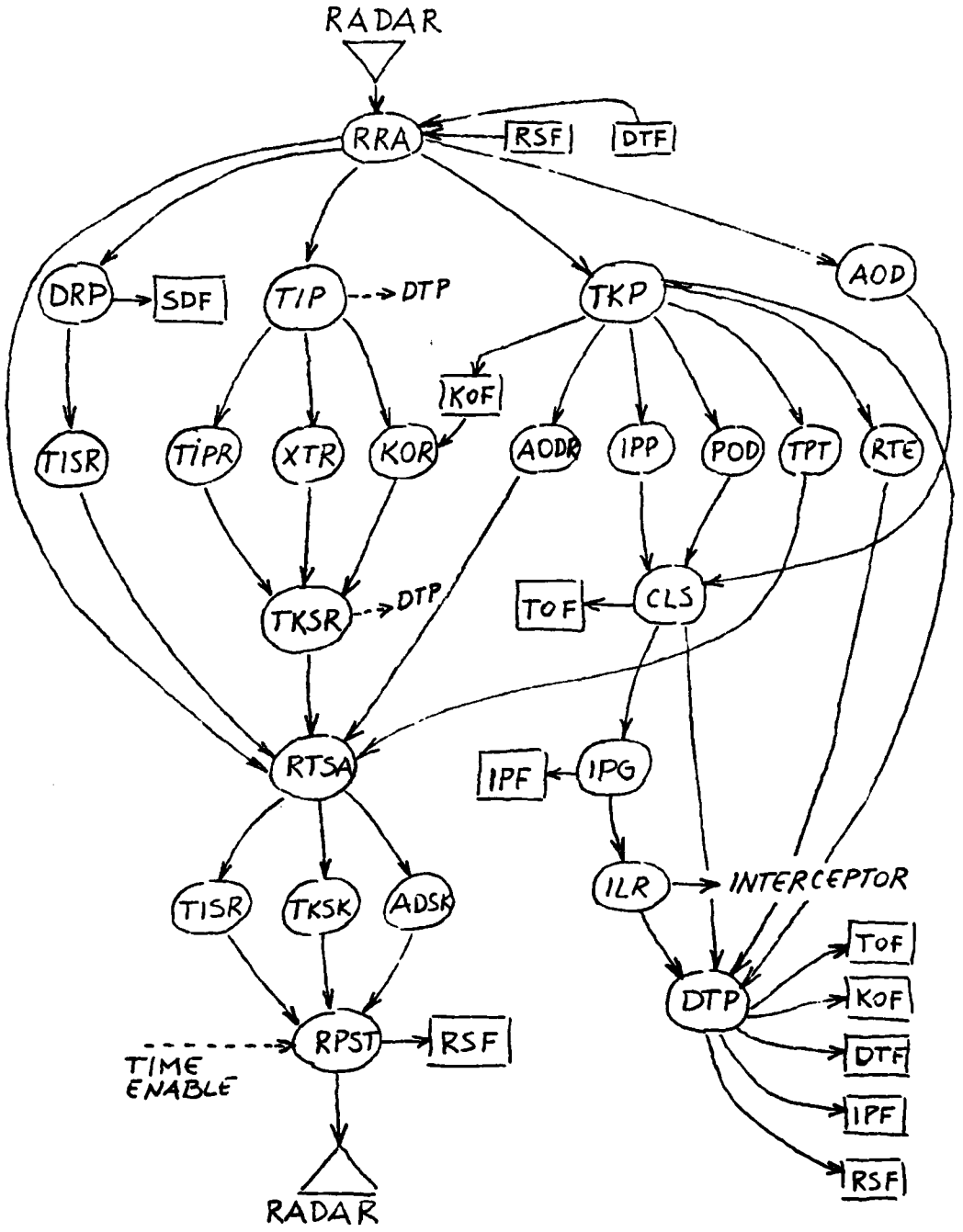


Figure 8: A Sample BMD Application Structure

Appendix B

THE GENERALIZED CUBE NETWORK

The Generalized Cube Network [Mill80] is a multistage network with topology equivalent with those of Staran, Indirect Binary n-Cube, Omega [Sieg79], and (f=2,s=2) banyan [Rath80] networks. The network has N inputs and N outputs (fig. 9.) and $m = \log_2 N$ stages. Each stage consists of a set of N (I/O) lines connected to N/2 interchange boxes. Each interchange box is two-input, two-output device. Lines are labeled as integers from 0 to N-1.

The connections in the network are based on the cube interconnection function. If $l_{m-1}.....l_1l_0$ is the binary representation of an arbitrary I/O line, then the m cube interconnection functions can be defined as:

$$cube_i(l_{m-1}.....l_1l_0) = l_{m-1}....l_{i+1}\overline{l}_i l_{i-1}.....l_1l_0$$

where $0 \leqslant i \leqslant m-1$ and $\overline{li}$ denotes complement of li. That means that the cubei interconnection function connects I/O lines whose labels (their binary representation) differ in

i-th nit position. The stage i of the generalized cube topology performs the cubei interconnection function.

Each interchange box can connect input lines to the output lines in one of the four possible ways (fig. 9.):

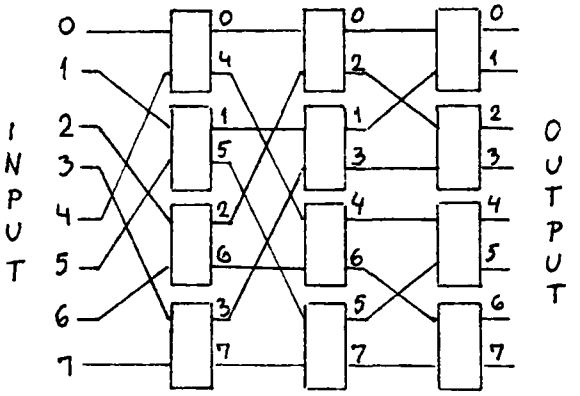
1. straight: input i to output i, input j to output j

2. exchange: input i to output j, input j to output i

3. lower broadcast: input j to both outputs

4. upper broadcast: input i to both outputs

A generalized cube network may have either the two-function interchange boxes (making connections 1. and 2.), or four function interchange boxes. Control of interchange boxes may be centralized (ie. performed by the same control unit), or decentralized, in which case each interchange box is controlled independently through the use of routing tags.



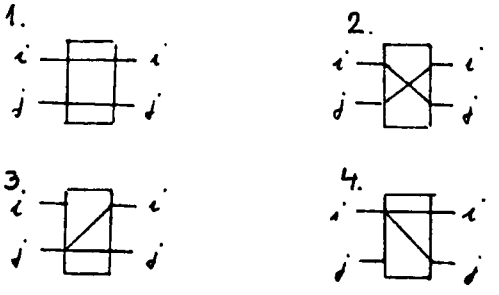


Figure 9: Cube Network with N = 8

Appendix C

PROCESS STATE MAINTENANCE AND KERNEL CALLS

A process (actor) is an instance of an act. An act is a reentrant segment of EPL code. Each process has a system-wide unique name. To provide for unique naming of processes in an environment in which processes are dynamically created and destroyed, process names are 32-bit long. The lowest order byte in the process name specifies the node where the process is resident. Once a process is destroyed, its name is never again reused as the name of a different process. Two words wide process name allows for up to 2^{24} different process names on each node of the distributed system.

The state information of a process is maintained by the Kernel within the process descriptor. The organization of a process descriptor is shown in fig. 10.

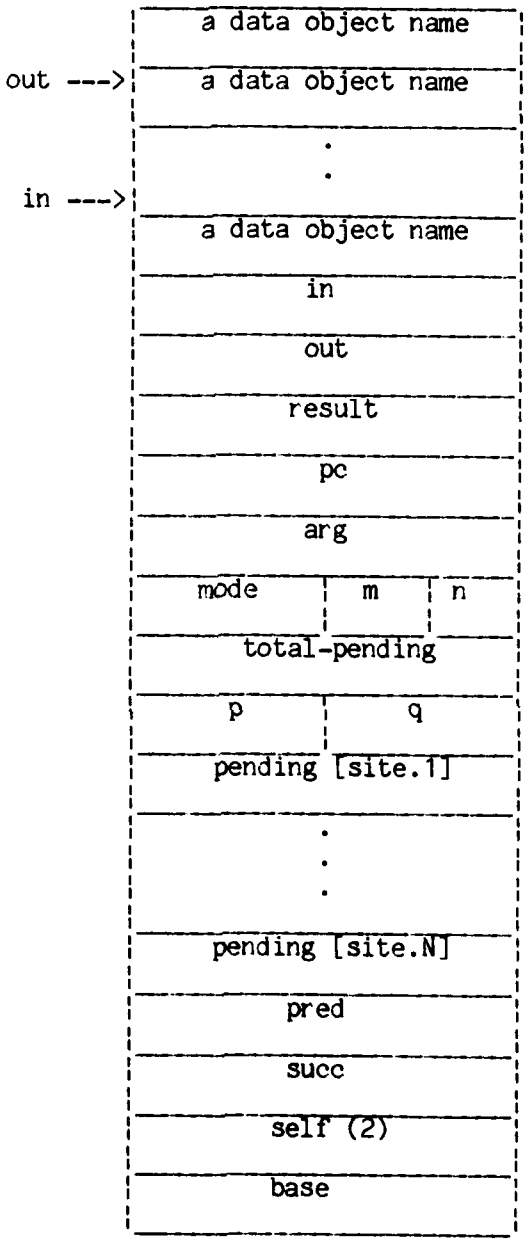


Figure 10: Process Descriptor Organization

in, out	- pointers to the circular buffer of data object names.
result	- result of the system call. The Kernel saves the result of the call (-1 - successful, 0 - unsuccessful) in the high order byte, and the Return Code in the low order byte.
pc	- program counter
arg	- address of arguments on the process's data segment.
mode	- in the case of interprocess communication, identifies passing of capabilities, as opposed to passing of messages.
m	- total number of processes receiving capabilities from this process, or sending capabilities to this process. (Total number of transactions.)
n	- nimber of outstending transactions
total-pending	- total number of processes waiting to send messages or capabilities to this process.
pending[site.i]	- total number of processes from site i waiting to send messages or capabilities to this process.
p, q	- pointers to array whose first element is pending[site.1], and last element pending[site.N]. Pointers p and q point to sites involved in the last execution of REC and RECC primitive, respectively. These two pointers allow for fair policy of selecting senders in the case of execution of next REC or RECC primitive.
pred, succ	- links to predecessor/successor in one of lists maintained by the Kernel or the Communication Subsystem ([Cohe81]).
self	- two words long name of the process.
base	- pointer to the base of the process data segment.

SENDC expr [exprlst] <delim expr [exprlst]>* <ONFAILURE stat>

Sends capabilities determined by evaluating expressions in each 'exprlst' to processes whose names are found by evaluating each expression 'expr'. The order in which receiving processes receive capabilities, is not dependant on the order in which receiving processes are named by the sending process. Maximum of 16 processes may be named by the sending process. Each 'expr' must evaluate into a valid process name. If the keyword ONFAILURE is specified, and the call fails, the statement or sequence of statements following ONFAILURE, is executed. If the ONFAILURE is not specified and the call fails, the calling process terminates. Execution of the sending process is suspended until all capabilities have been received.

If, following the call, the value of the register r1 is -1, the call was successful and the calling process expects no arguments to be placed on its data segment.

If, following the call, the value of the register r1 is 0, the call was unsuccessful and the calling process expects Return Code at the top of its data segment. The Return Code consists of two words; the first word specifies the reason for unsuccessful termination of the call, the second word identifies the receiving process that caused unsuccessful

termination of the call. (The process is identified by specifying its position in sending process's list of receivers.)

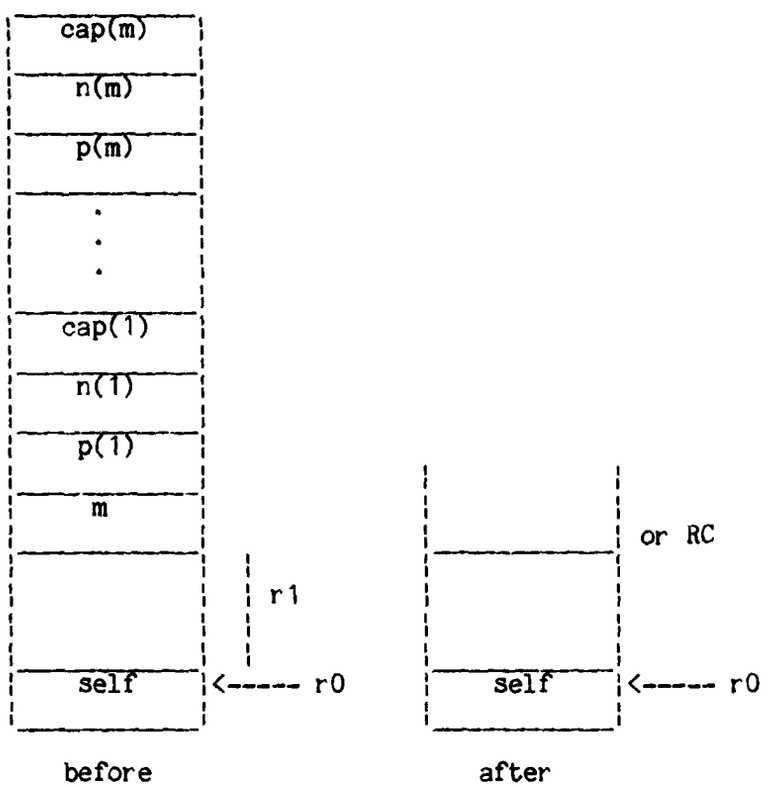


Figure 11: Data Segment for the process executing SENDC

m - number of receiving processes.

p(i) - name of i'th process in the list of receiving processes.

n(i) - number of capabilities to be sent to the i'th process.

cap(i) - capabilities to be sent to the i'th process.

Return Codes:

RC(1) = 1 receiving process identified by RC(2) is nonexistant.

3 receiving process identified by RC(2) is in TERMINATED state.

5 receiving process identified by RC(2) is in different mode from sender.

7 the number of capabilities specified by the sender, and the number of capabilities specified by the receiver identified by RC(2), are different.

RECFC expr [exprlst] <delim expr [exprlst]>* <ONFAILURE stat>

Receives capabilities from processes whose names are found by evaluating each expression 'expr', and assignes them to variables specified by 'exprlst'. The order in which sending processes send capabilities, is not dependent on the order in which sending processes are named by the receiving process. Maximum of 16 processes may be named by the receiving process. Each 'expr' must evaluate into a valid process name. If the keyword ONFAILURE is specified, and the call fails, the statement or sequence of statements following ONFAILURE, is executed. If the ONFAILURE is not specified and the call fails, the calling process terminates. Execution of the calling process is suspended until all capabilities have been received.

If, following the call, the value of the register r1 is -1, the call was successful and the calling process expects capabilities to be placed in reserved spaces on its data segment.

If, following the call, the value of the register r1 is 0, the call was unsuccessful and the calling process expects Return Code at its data segment. The Return Code consists of two words; the first word specifies the reason for unsuccessful termination of the call, the second word

identifies the sending process that caused unsuccessful termination of the call. (The process is identified by specifying its position in receiving process's list of senders.)

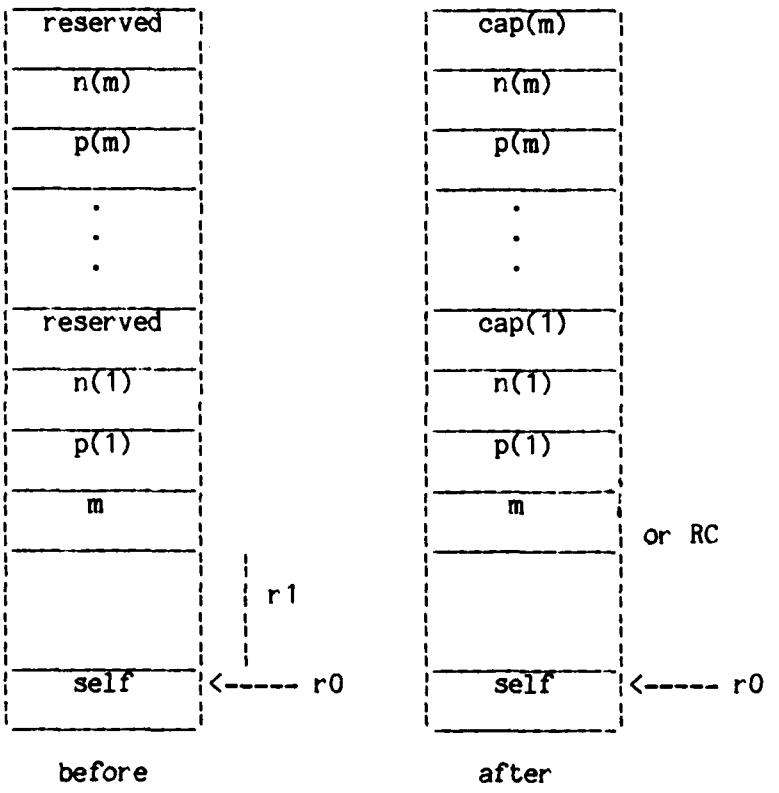


Figure 12: Data Segment for the process executing RECFC

m - number of sending processes.

p(i) - name of i'th process in the list of sending processes.

n(i) - number of capabilities expected to be received from i'th process.

reserved - space reserved for capabilities.

cap(i) - capabilities received from i'th process.

Return Codes:

RC(1) = 1 sending process identified by RC(2) is nonexistant.

3 sending process identified by RC(2) is in TERMINATED state.

5 sending process identified by RC(2) is in different mode from receiver.

7 the number of capabilities specified by the receiver, and the number of capabilities specified by the sender identified by RC(2), are different.

RECC expr [exprlst] <delim expr [exprlst]>* <ONFAILURE stat>

Receives capabilities from one of processes whose names are found by evaluating each expression 'expr'. Capabilities are assigned to variables specified by 'exprlst'. Maximum of 16 processes may be named by the receiving process. Each 'expr' must evaluate into a valid process name. If the keyword ONFAILURE is specified, and the call fails, the statement or sequence of statements following ONFAILURE, is executed. If the ONFAILURE is not specified and the call fails, the calling process terminates. Execution of the calling process is suspended until capabilities from one of specified senders have been received.

If, following the call, the value of the register r1 is -1, the call was successful and the calling process expects name of the sender and capabilities to be placed at the top of its data segment.

If, following the call, the value of the register r1 is 0, the call was unsuccessful and the calling process expects Return Code at its data segment. Return Code consists of two words; the first word specifies the reason for unsuccessful termination of the call, the second word identifies the sending process that caused unsuccessful termination of the

call. (The process is identified by specifying its position in receiving process's list of candidate senders.)

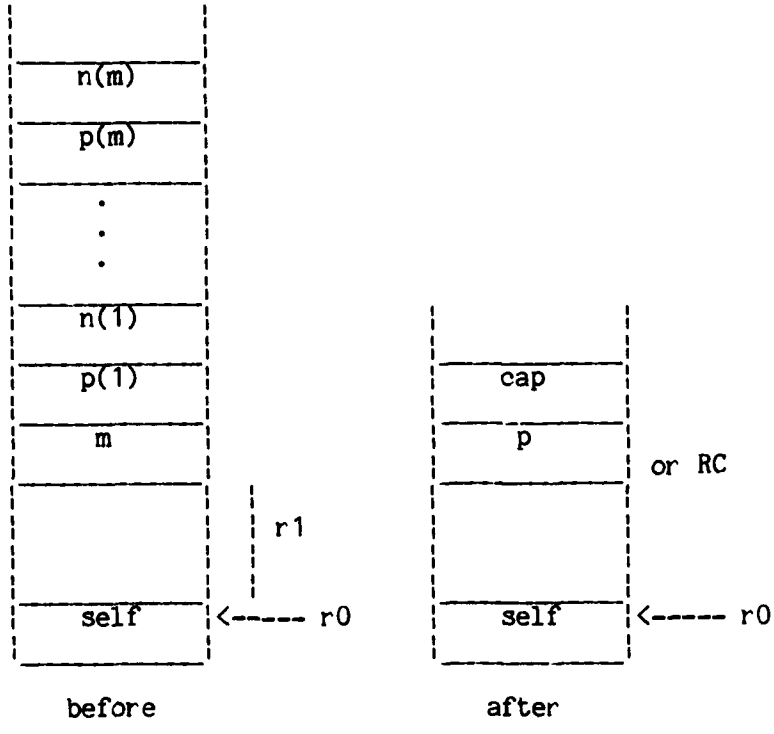


Figure 13: Data Segment for the process executing RECC

m - number of candidate senders.

p(i) - name of i'th candidate sender.

n(i) - number of capabilities expected to be received from i'th candidate sender.

p - name of the selected sender.

cap - capabilities received from selected sender.

Return Codes:

RC(1) = 1 candidate sender identified by RC(2) is nonexistant.

3 candidate sender identified by RC(2) is in TERMINATED state.

5 candidate sender identified by RC(2), and receiver are in different modes.

7 the number of capabilities specified by the receiver, and the number of capabilities specified by the candidate sender identified by RC(2), are different.

Kernel to Kernel protocols and message formats

Create Request:

CREATE.1 - Kernel entry that receives CREATE requests and creates an instance of an EPL act on the local CPU. It receives messages in the following format:

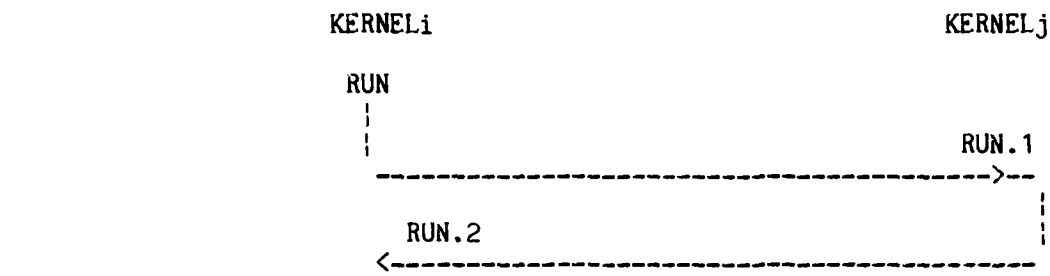
```
FUNCTION NAME (1) ====> CREATE.1
ACT NAME (2)
NAME OF CREATING PROCESS (2)
```

CREATE.2 - Kernel entry that receives the child name for the parent process. It receives messages in the following format:

```
FUNCTION NAME (1) ====> CREATE.2
NAME OF THE NEW PROCESS (2)
NAME OF CREATING PROCESS (2)
```

Numbers in parenthases denote the number of 16-bit words per item. When the I/O subsystem on the CPU side of the destination node receives a message, it executes procedure call to the Kernel entry specified in the FUNCTION NAME.

Run Request:



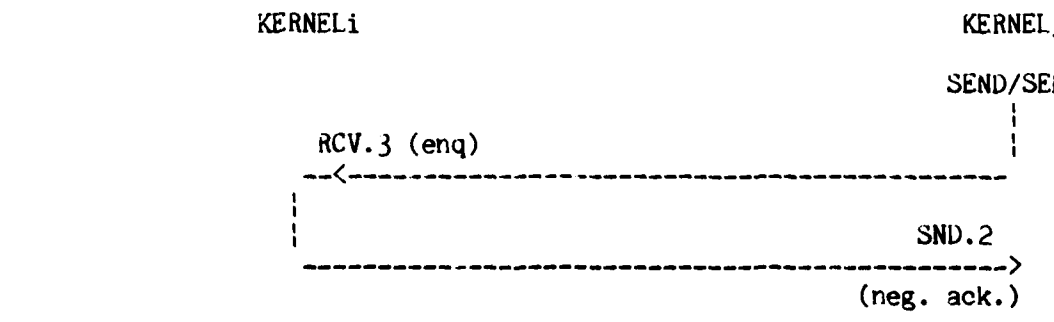
RUN.1 - Kernel entry that receives initialization arguments for the newly created process. It receives messages in the following format:

```
FUNCTION NAME (1) ====> RUN.1
CHILD NAME (2)
PARENT NAME (2)
NUMBER OF ARGUMENTS (1)
ARG(1)  (1)
  .
  .
ARG(k)  (1)
```

RUN.2 - Kernel entry that receives positive acknowledgement following the RUN request. Positive acknowledgements have the following format:

```
FUNCTION NAME (1) ====> RUN.2
CHILD NAME (2)
PARENT NAME (2)
```

Send First:

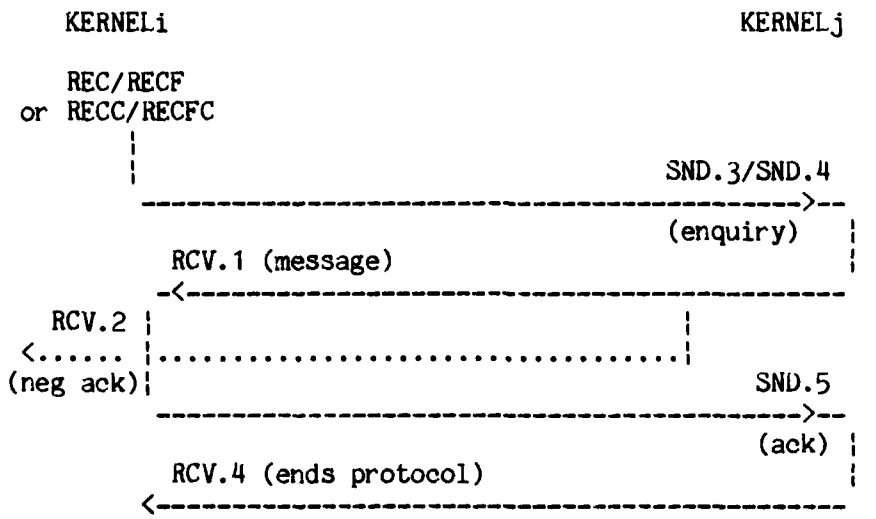


RCV.3 - Kernel entry that receives enquiries from senders. It receives messages in the following format:

```
FUNCTION NAME (1) ====> RCV.3
MODE (1)
NAME OF THE SENDER (2)
NAME OF THE RECEIVER (2)
```

SND.2 - Kernel entry that receives negative acknowledgements from candidate receivers. Negative acknowledgements have the following format:

```
FUNCTION NAME (1) ====> SND.2
NAME OF THE SENDER (2)
NAME OF THE RECEIVER (2)
RETURN CODE (1)
```



SND.3 - Kernel entry that receives enquiries from receiving processes executing REC or RECC primitive. It receives messages in the following format:

```
FUNCTION NAME (1) ====> SND.3
MODE (1)
NAME OF THE SENDER (2)
```

NAME OF THE RECEIVER (2)

SND.4 - Kernel entry that receives enquiries from receiving processes executing RECF or RECFC primitive. It receives messages in the following format:

```
FUNCTION NAME (1) ====> SND.4
MODE (1)
NAME OF THE SENDER (2)
NAME OF THE RECEIVER (2)
```

RCV.1 - Kernel entry that receives message or capabilities from a sender for a receiver, in the following format:

```
FUNCTION NAME (1) ====> RCV.1
NAME OF THE SENDER (2)
NAME OF THE RECEIVER (2)
NUMBER OF ARGUMENTS (1)
ARG(1)   (1)
  .
  .
ARG(k)   (1)
```

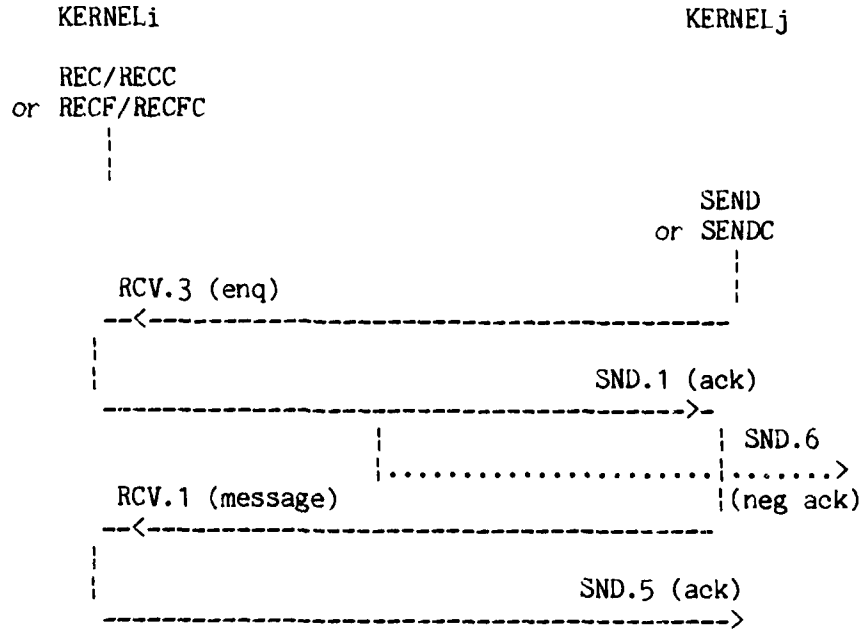
RCV.2 - Kernel entry that receives negative acknowledgement from sender's site, for reason specified by Return Code. Format of negative acknowledgements was described with entry SND.2. (The difference of course, is that FUNCTION NAME in this case contains the name of the entry RCV.2.)

SND.5 - Kernel entry that receives positive acknowledgement from receiver after it received a message or capabilities.

RCV.4 - Kernel entry that receives positive acknowledgement which ends protocol. Format of the positive acknowledgement is:

FUNCTION NAME (1)
NAME OF THE SENDER (2)
NAME OF THE RECEIVER (2)

Receive first with no pending senders:

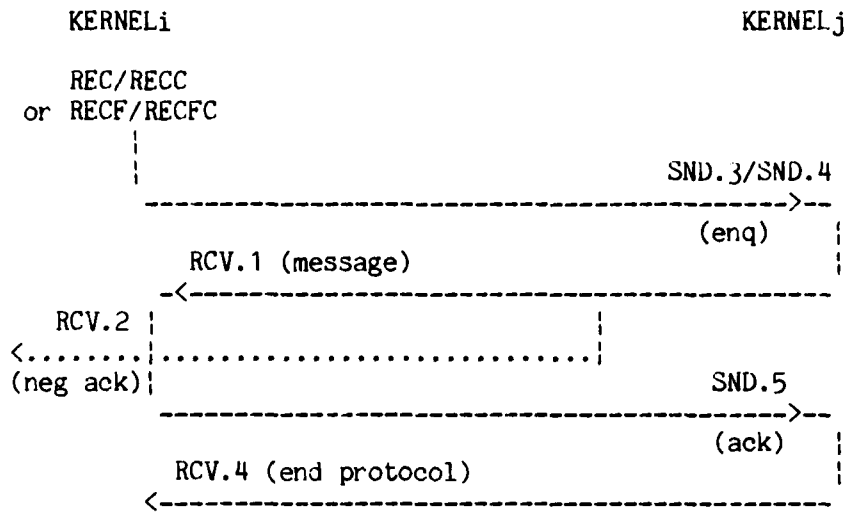


SND.1 - Kernel entry that receives positive acknowledgement from receiver's site. Format of positive acknowledgement was previously defined.

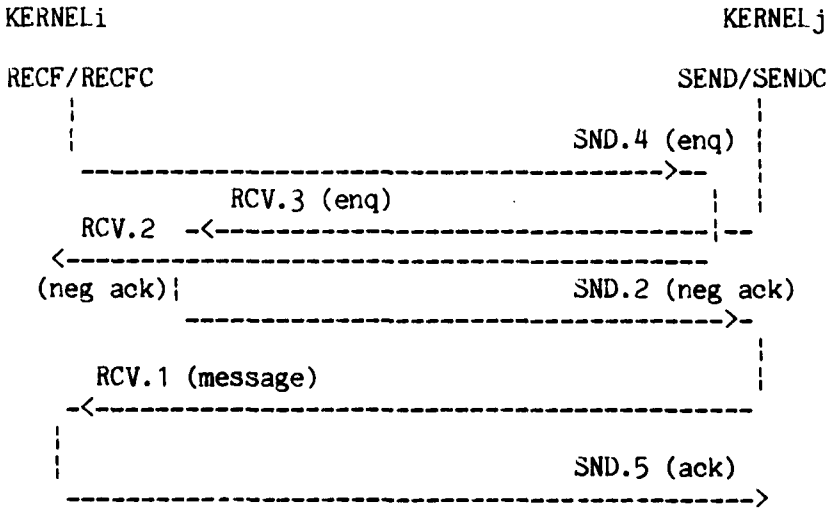
SND.6 - Kernel's entry that receives negative acknovledgements. Reason for negative acknovledgement is specified by Return Code.

RCV.2 - Kernel's entry that receives negative acknowlegement for the reason specified by the Return Code (sender and receiver do not name each other, sender does not exist or is in TERMINATED state, sender and receiver are in different mode).

Receive first with pending senders:



SEND and RECEIVE concurrently with pending senders:

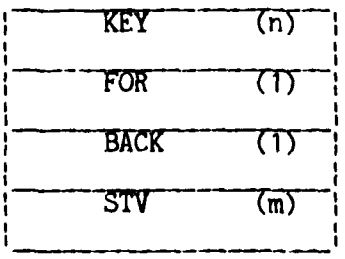


Appendix D

THE FILE MANAGEMENT CALLS AND DATA STRUCTURES

A file is a contiguous segment of global memory with the system-wide unique name. The name of a file is 16-bits long; low order byte specifies the node where the file is resident, seven low order bits in the high order byte specify one of 128 possible Segment Descriptor Registers of the local pair of Memory Management Units, and the highest order bit in the file name identifies the file segment as oposed to a data segment. The Segment Descriptor Register identifies the base, extent, and attributes of the segment.

Each file is organized as a double linked list of elements of the same size. Each element of a file consists of a KEY which is an unique identifier of the element within that file, and the State Vector which describes the current state of that element. The file element is organized as follows:



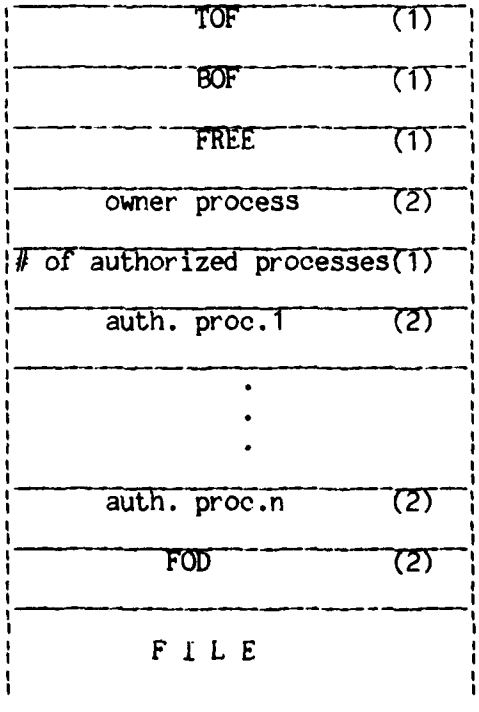
KEY: n-word long, unique identifier of the file element

FOR: forward list pointer

BACK: backward list pointer

STV: m-words long State Vector that describes the current state of the element in the Problem Space.

The current state of a file is maintained by the file menager within the File Descriptor. The File Descriptor specifies the organization of the file and names processes authorized to access the file in the read-write mode. The number of authorized processes is specified by the owner process during the file creation and can not be changed afterwards. The File Descriptor is organized as follows:



TOF: pointer to the beginning of the forward list of elements.(Top of the file).

BOF: pointer to the beginning of the backward list of elements.(Bottom of the file).

FREE: pointer to the list of free space.

FOD: File Organization Descriptor. The first word specifies the number of words in the KEY; the second word specifies the number of words in the State Vector.

ALLOCF expr [exprlst] <ONFAILURE stat>

Allocates contiguous segment of global memory of the same node where the calling process resides, and associates a system-wide unique name with the segment.

expr: returns the name of the file

exprlst: specifies the number of 128-word blocks to be allocated, authorized processes, and the File Organization Descriptor.

stat: statement or a sequence of statements to be executed in the case of unsuccessfull execution of the primitive.

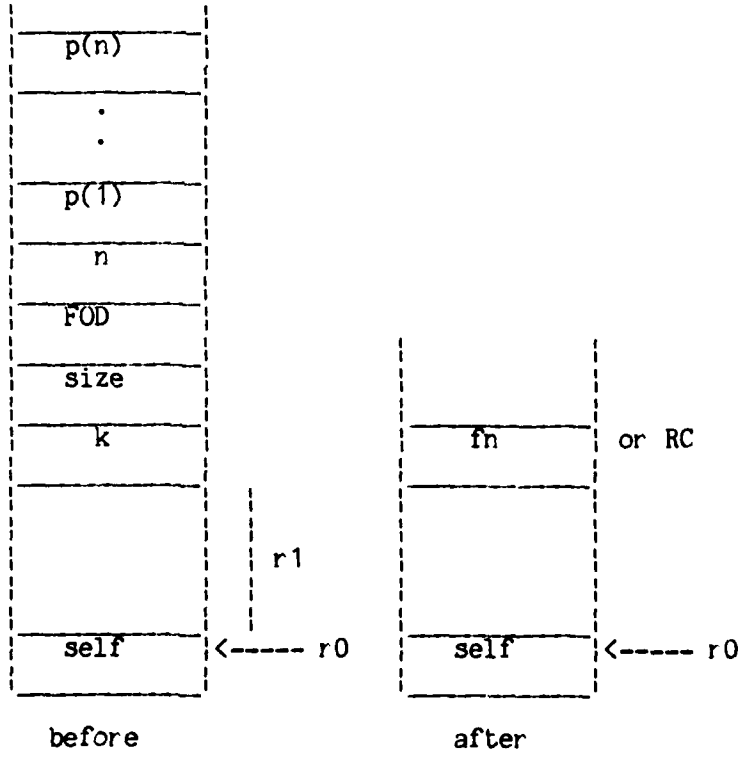


Figure 14: Data Segment for the process executing ALLOCF

k - number of arguments on the process's stack

n - number of authorized processes

FOD - File Organization Descriptor

p(i)- i'th authorized process

fn - file name

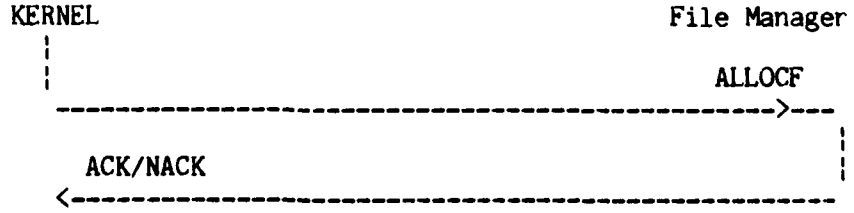
If, following the call, the value of the register r1 is -1, the call was successfull and the calling process expects the name of the file on the top of its data segment.

If, following the call, the value of the register r1 is 0, the call was unsuccessfull and the calling process expects Return Code at the top of its data segment.

Return Codes:

RC = 1 indicates that there is no space on the native node

Kernel - File Manager protocol (messages are exchanged between CPU and FPU of the same node):



Format of the message sent by Kernel and received by the File Manager:

```
FUNCTION NAME (1)  ====> ALLOCF
CALLING PROCESS (2)
FILE ORGANIZATION DESCRIPTOR (1)
NUMBER OF AUTHORIZED PROCESSES (1)
AUTHORIZED PROCESS1  (2)
        .
        .
AUTHORIZED PROCESSn  (2)
```

The FUNCTION NAME contains the name of File Manager's entry (ALLOCF) to be called upon the receipt of the message.

Format of the message sent by the File Manager and received by Kernel's entry ACK or NACK:

```
FUNCTION NAME (1)  =====> ACK
CALLING PROCESS (2)
FILE NAME (1)
```

```
FUNCTION NAME (1)  =====> NACK
CALLING PROCESS (2)
RETURN CODE (1)
```

The FUNCTION NAME contains the name of Kernel's entry (ACK or NACK) to be called upon the receipt of the message. Note that in this case the same entry may be used to pass the message to the process, because the Kernel does not interpret messages and has no knowledge wheather it contains the file name or the Return Code. For practical reasons, during coding, it is possible to combine entries so that the

Kernel, Memory manager, and File Manager have much smaller number of entries than it appears here. For the design step, though, clarity is more important than bravity, so each entry is separately specified.

DEL expr [exprlst] <ONFAILURE stat>

Deletes an element of the file specified by 'expr'.

expr: must evaluate into a file name

exprlst: specifies the KEY of the element to be deleted from the file.

stat: statement or a sequence of statements to be executed in the case of unsuccessfull execution of the primitive.

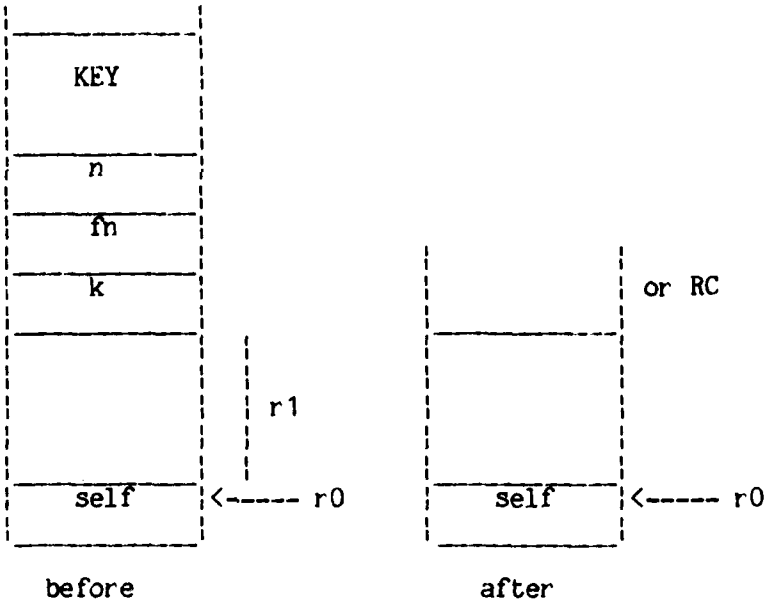


Figure 15: Data Segment for the process executing DEL

k - number of arguments on the process's stack

fn - file name

n - number of words in the KEY

If, following the call, the value of the register r1 is -1, the call was successfull and the calling process expects no arguments returned by the File Manager.

If, following the call, the value of the register r1 is 0, the call was unsuccessfull and the calling process expects Return Code at the top of its data segment.

Return Codes:

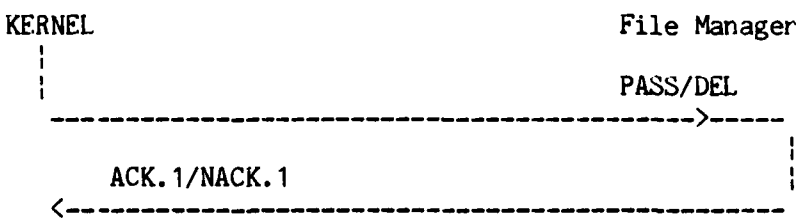
RC = 1 reference to a nonexistant file

3 unauthorized access

5 number of words in the KEY differs from what is specified in the File Organization Descriptor.

7 the Key not found.

Kernel - File Manager protocol:



The Kernel examines the name of the file, and if the file is coresident, it calls the File Manager's entry DEL. If the file resides on different node, the Kernel calls the Network Interface entry PASS. The Network Interface does not interpret the message; it expects that the second word of the message denotes the destination site, and that the third

word of the message contains the number of words to be sent to the destination site. On the call to this entry, the Network Interface calculates the routing tag and writes the message into the output FIFO buffer. In this case the protocol involves the Kernal and the File Manager from different sites.

Format of the message sent by the Kernel and received by the File Manager:

```
FUNCTION NAME (1) =====> DEL
CALLING PROCESS (2)
FILE NAME (1)
NUMBER OF WORDS/KEY (1)
THE KEY (n)
```

The FUNCTION NAME contains the name of the File Manager's entry (DEL) to be called upon the receipt of the message.

Format of the message sent by the File Manager and received by the Kernel's entry ACK.1 or NACK.1:

```
FUNCTION NAME (1) =====> ACK.1
CALLING PROCES (2)

FUNCTION NAME (1) =====> NACK.1
CALLING PROCESS (2)
RETURN CODE (1)
```

ADD expr [exprlst] <ONFAILURE stat>

Adds an element to the end of the forward list of elements of the file named by 'expr', and returns the KEY to the calling process.

expr: must evaluate into a file name

exprlst: specifies the State Vector and the size of the KEY of the element to be added to the file.

stat: statement or a sequence of statements to be executed in the case of unsuccessfull execution of the primitive.

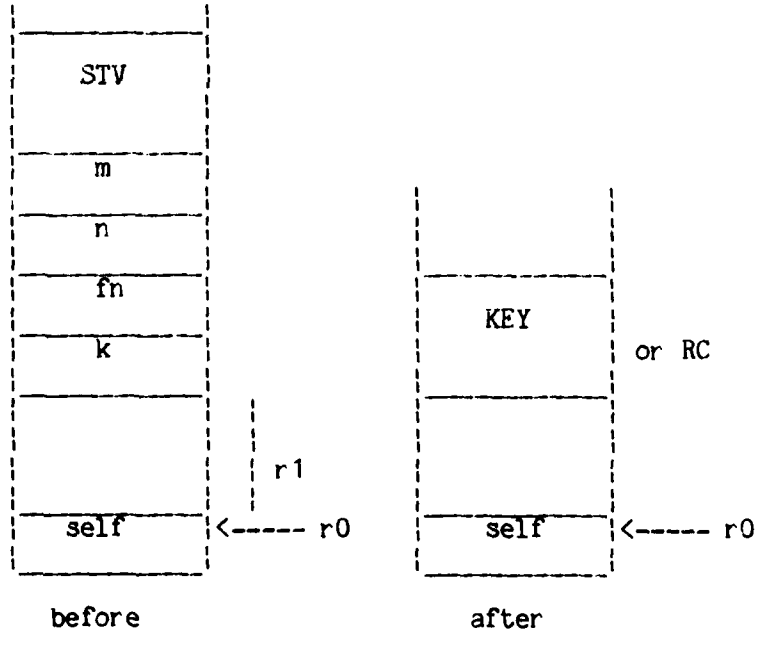


Figure 16: Data Segment for the process executing ADD

k - number of arguments on the process's stack

fn - file name

n - number of words in the KEY

m - number of words in the State Vector (STV)

If, following the call, the value of the register r1 is -1, the call was successfull, and the calling process expects no arguments returned by the File Manager.

If, following the call, the value of the register r1 is 0, the call was unsuccessfull and the calling process expects Return Code at the top of its data segment.

Return Codes:

RC = 1 reference to a nonexistant file

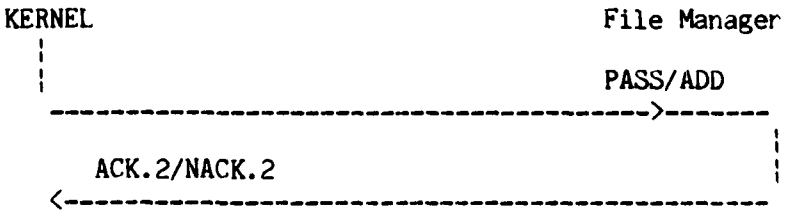
3 unauthorized access

5 number of words in the KEY differs from what is specified in the File Organization Descriptor.

7 number of words in the State Vector differs from what is specified in the File Organization Descriptor.

9 no space available in the file

Kernel - File Manager protocol:



The Network Interface entry PASS was described earlier. Format of the message sent by the Kernel and received by the File Manager:

```
FUNCTION NAME (1) =====> ADD
CALLING PROCESS (2)
NUMBER OF WORDS/KEY (1)
NUMBER OF WORDS/STV (1)
THE STV (m)
```

Format of the message sent by the File Manager and received by Kernel's entry ACK.2 or NACK.2:

```
FUNCTION NAME (1) =====> ACK.2
CALLING PROCESS (2)
NUMBER OF WORDS/KEY (1)
THE KEY (n)
```

```
FUNCTION NAME (1) =====> NACK.2
CALLING PROCESS (2)
RETURN CODE (1)
```

UPDATE expr [exprlst] <ONFAILURE stat>

Updates the State Vector of the file element with the specified KEY.

expr: must evaluate into a file name

exprlst: specifies the KEY and the new State Vector of the file element that is to be updated

stat: statement or a sequence of statements to be executed in the case of unsuccessfull execution of the primitive.

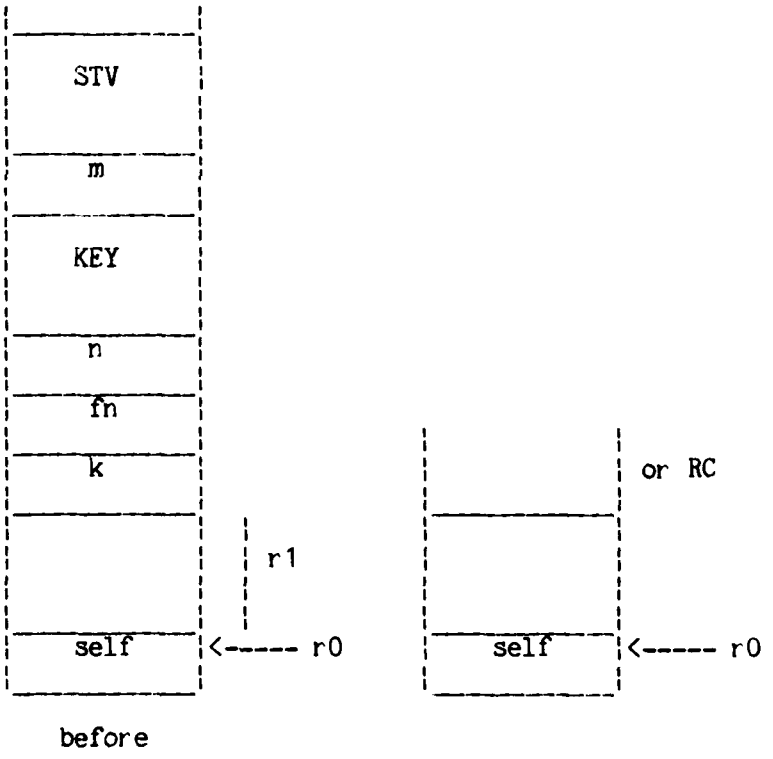


Figure 17: Data Segment for the process executing UPDATE

k - number of arguments on the process's stack

fn - file name

n - number of words in the KEY

m - number of words in the State Vector (STV)

If, following the call, the value of the register r1 is -1, the call was successfull, and the calling process expects no arguments returned by the File Manager.

If following the call, the value of the register r1 is 0, the call was unsuccessfull and the calling process expects Return Code at the top of its data segment.

Return Codes:

RC = 1 reference to a nonexistant file

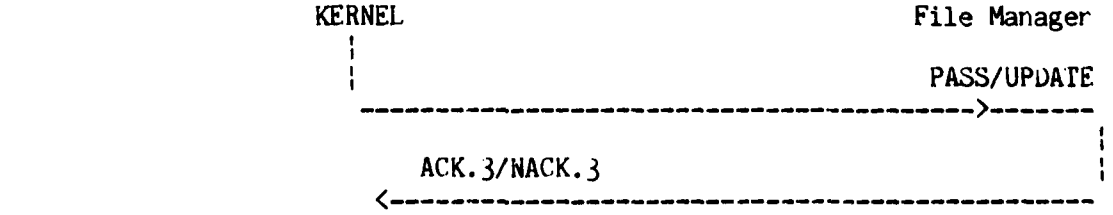
3 unauthorized access

5 number of words in the KEY differs from what is specified in the File Organization Descriptor.

7 number of words in the State Vector differs from what is specified in the File Organization Descriptor.

9 the KEY not found

Kernel - File Manager protocol:



The Network Interface entry PASS is described with the primitive DEL. Format of the message sent by the KERNEL and received by the File Manager:

```
FUNCTION NAME (1) =====> UPDATE
CALLING PROCESS (2)
FILE NAME (1)
NUMBER OF WORDS/KEY (1)
THE KEY (n)
NUMBER OF WORDS/STV (1)
THE STV (m)
```

Format of the message sent by the File Manager and received by the Kernel's entry ACK.3 or NACK.3:

```
FUNCTION NAME (1) =====> ACK.3
CALLING PROCESS (2)
```

```
FUNCTION NAME (1) =====> NNACK.3
CALLING PROCESS (2)
RETURN CODE (1)
```

COPY expr [exprlst] <ONFAILURE stat>

Copies the specified KEY, State Vector, or entire element of the file named by 'expr'.

expr: must evaluate into a file name

exprlst: must evaluate into two parameters (denoted PARM1 and PARM2)

stat: statement or a sequence of statements to be executed in the case of unsuccessfull execution of the primitive.

PARM1 specifies direction of the movement through the file:

FOR - specifies forward movement through the file starting with the currently first element of the file. The File Manager creates the File Access Control Table (FACT) for the calling process, first time it executes this primitive.

BACK - specifies backward movement through the file starting with the presently last element of the file. The File Manager creates FACT for the calling process, first time the primitive is executed.

END - terminates the movement through the file. The FACT for the calling process is destroyed.

LAST - specifies that only the last element of the file is to be copied. The FACT for the calling process is not created.

KEY - specifies that the State Vector of the element with the specific key is to be copied. The FACT for the calling process is not created.

The File Access Control Table for a process, maintains the relative position of the process within the file during its scan through the file, in the presence of other processes having concurrent read-write access to the same file. For more details about the FACT see section 4.3.

PARM2 designates the portion of the file element, specified by PARM1, that is to be copied for the calling process:

KEY - only the KEY is to be copied and sent to the calling process.

STV - only the State Vector is to be copied and sent to the calling process.

EL - the whole element is to be copied and sent to the calling process.

If, following the call, the value of the register r1 is -1, the call was successfull, and the calling process expects arguments (the KEY, the State Vector, or the whole element of the file) to be placed on its data segment by the File Manager.

If, following the call, the value of the register r1 is 0, the call was unsuccessfull and the calling process expects the Return Code at its data segment.

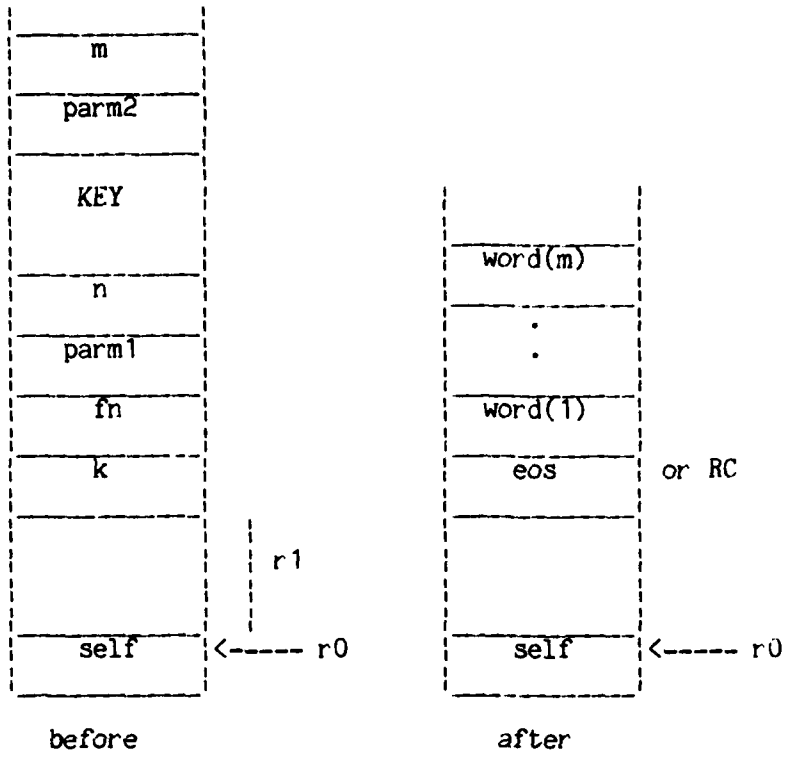


Figure 18: Data Segment for the process executing COPY

k - number of words on the process's stack

fn - file name

n - number of words in the KEY (significant only if PARM1 = KEY; if not, n and KEY are not present on the process's data segment).

m - number of words in the item specified by PARM2.

eos- end of scan indicator

Return Codes:

RC = 1 reference to a nonexistant file

3 FACT did not exist for the calling process (the process terminates nonexisting scan).

5 movement inconsistent with the direction of scan specified in FACT.

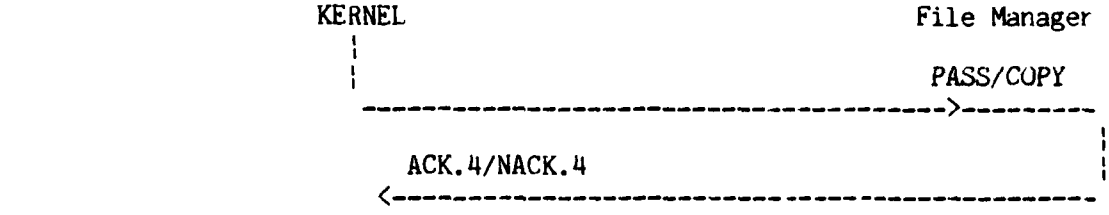
7 number of words in the KEY differs from what is specified in the File Organization Descriptor.

9 number of words in the State Vector differs from what is specified in the File Organization Descriptor.

11 number of words in the file element differs from what is specified in the File Organization Descriptor.

13 the key not found

Kernel - File Manager protocol:



Format of the message sent by the Kernel and received by File Manager:

```
                FUNCTION NAME (1) =====> COPY
                CALLING PROCESS (2)
                FILE NAME (1)
                PARM1
optional -->    NUMBER OF WORDS/KEY (1)
optional -->    THE KEY (n)
                PARM2 (1)
                NUMBER OF WORDS/ITEM
```

Format of the message sent by the File Manager and received by the Kernel's entry ACK.4 or NACK.4 :

```
FUNCTION NAME (1) =====> ACK.4
CALLING PROCESS (2)
NUMBER OF WORDS (1)
WORD(1) (1)
   .
   .
WORD(m) (1)


FUNCTION NAME (1) =====> NACK.4
CALLING PROCESS (2)
RETURN CODE (1)
```

Appendix E

MEMORY MANAGER CALLS

ALLOC expr [exprlst] <ONFAILURE stat>

Allocates contiguous segment of global memory (data object) for the calling process. Expression list 'exprlst' specifies the number of 128-word blocks to be allocated for the data object; maximum number of blocks to be allocated is 256. The capability for the data object is returned and associated with variable specified by expression 'expr'. A data object name is 16-bit long; the low order byte specifies the node where the data object is resident, seven low order bits in the higher order byte specify one of 128 Segment Descriptor Registers of the local pair of Memory Management Units, and the highest order bit in the object name identifies the data segment as opposed to a file segment. The Segment Descriptor Register identifies the base, extent, and attributes of the segment.

In the case of unsuccessful execution of the primitive, the statement or sequense of statements following the keyword ONFAILURE is executed. If the ONFAILURE is not specified and the call fails, the calling process terminates.

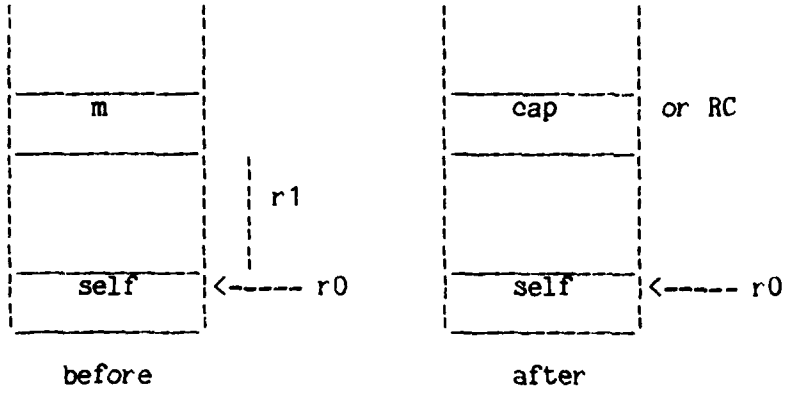


Figure 19: Data Segment for the process executing ALLOC

m - number of 128-word blocks to be allocated

cap - capability for the data object

The Memory Manager attempts to allocate the segment of global memory of the same node where the calling process is resident. If node K can not allocate the space, it forwards the request to the node (K+1) mod N where N is the total number of nodes in the system. If the allocation request reaches the original node, negative acknowledgement is sent back to the Kernel.

If, following the call, the value of the register r1 is -1, the call was successful and the calling process expects capability for created object to be placed on its data segment.

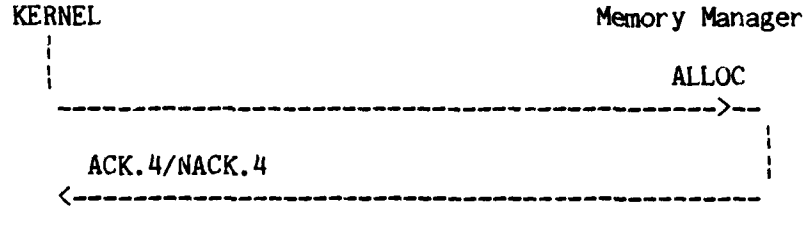
If, following the call, the value of the register r1 is 0, the call was unsuccessful, and the calling process expects Return Code to be placed on its data segment.

Return Codes:

RC = 1 the number of blocks requested exceeds the maximum allowable number of blocks

3 no space available

Kernel - Memory Manager protocol:



Format of the message sent by the Kernel and received by Memory Manager's entry ALLOC:

FUNCTION NAME (1) =====> ALLOC
CALLING PROCESS (2)
DATA OBJECT SIZE (1)

Format of the message sent by the Memory Manager and received by Kernel's entry ACK.4 or NACK.4 :

FUNCTION NAME (1) =====> ACK.4
CALLING PROCESS (2)
DATA OBJECT NAME (1)

FUNCTION NAME (1) =====> NACK.4
CALLING PROCESS (2)
RETURN CODE (1)

FREE expr <ONFAILURE stat>

Destroyes a data object and returns the corresponding segment of global memory to the pool of unused memory space. Expression 'expr' specifies capability that points to the data object to be destroyed. In the case of unsuccessful execution of the primitive, the statement or sequense of statements following the keyword ONFAILURE is executed. If the ONFAILURE is not specified and the call fails, the calling process terminates.

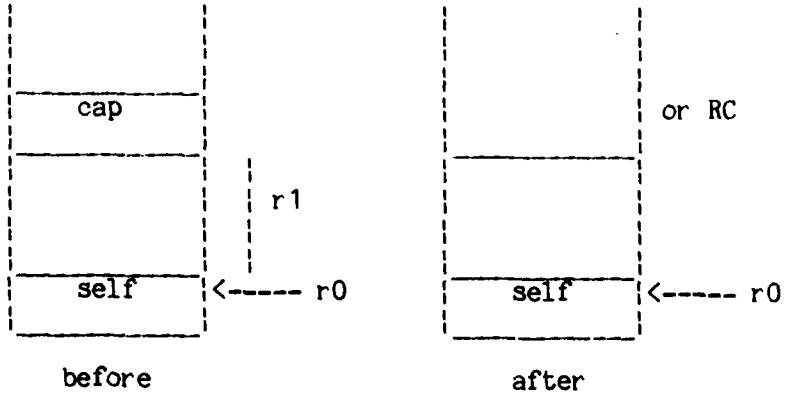


Figure 20: Data Segment for the process executing FREE

cap - capability for the data object

If, following the call, the value of the register r1 is -1, the call was successful and the calling process expects no arguments to be placed on its data segment.

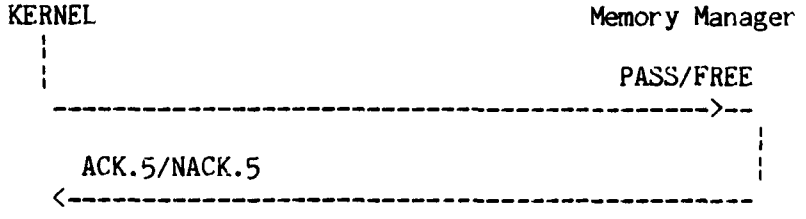
If, following the call, the value of the register r1 is 0, the call was unsuccessful, and the calling process expects Return Code to be placed on its data segment.

Return Codes:

RC = 1 nonexisting capability specified by the calling process

3 the capability specified by the calling process maps into invalid object name

Kernel - Memory Manager protocol:



Format of the message sent by the Kernel and received by Memory Manager's entry FREE:

FUNCTION NAME (1) =====> FREE
CALLING PROCESS (2)
OBJECT NAME (1)

Format of the message sent by the Memory Manager and received by Kernel's entry ACK.5 or NACK.5 :

```
FUNCTION NAME (1) =====> ACK.5
CALLING PROCESS (2)
```

```
FUNCTION NAME (1) =====> NACK.5
CALLING PROCESS (2)
RETURN CODE (1)
```

READ expr [exprlst] <ONFAILURE stat>

Reads a number of conseccutive words from a data object. Expression 'expr' must evaluate into capability that points to the data object. Expression list 'exprlst' specifies offset within the object and the number of words to be read. In the case of unsuccessful execution of the primitive, the statement or sequense of statements following the keyword ONFAILURE is executed. If the ONFAILURE is not specified and the call fails, the calling process terminates.

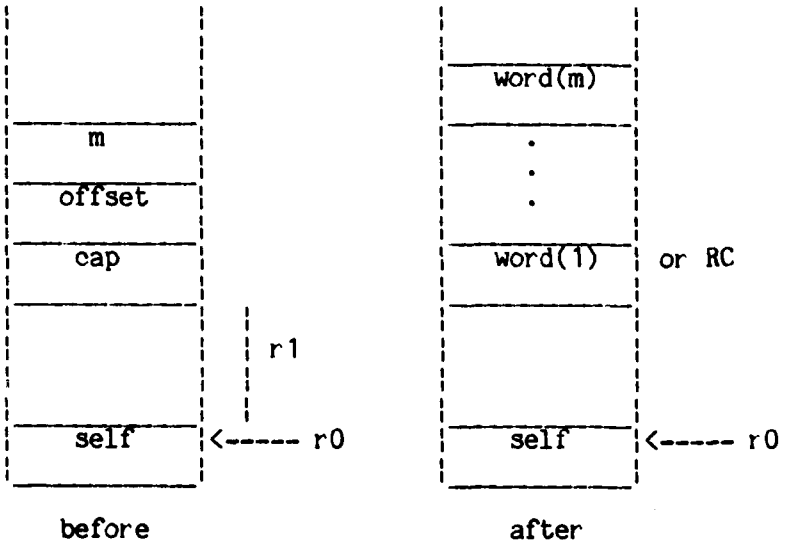


Figure 21: Data Segment for the process executing READ

cap - capability for the data object

m - number of words to be read

If, following the call, the value of the register r1 is -1, the call was successful and the calling process expects m words to be placed on its data segment.

If, following the call, the value of the register r1 is 0, the call was unsuccessful, and the calling process expects Return Code to be placed on its data segment.

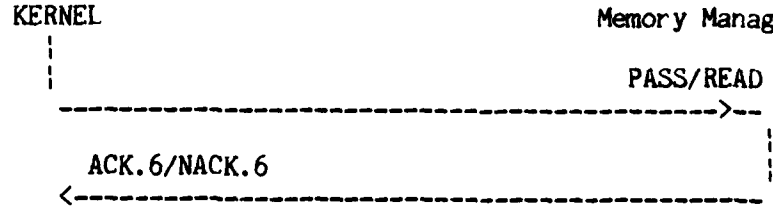
Return Codes:

RC = 1 nonexisting capability specified by the calling process

3 the capability specified by the calling process maps into invalid object name

5 segmentation exception - the caling process tries to read beoynd the limit specified by the limit field of the corresponding Segment Descriptor Register

Kernel - Memory Manager protocol:



Format of the message sent by the Kernel and received by Memory Manager's entry READ:

FUNCTION NAME (1) =====> READ
CALLING PROCESS (2)

```
OBJECT NAME (1)
OFFSET (1)
NUMBER OF WORDS (1)
```

Format of the message sent by the Memory Manager and received by Kernel's entry ACK.6 or NACK.6 :

```
FUNCTION NAME (1) =====> ACK.6
CALLING PROCESS (2)
NUMBER OF WORDS (1)
WORD(1) (1)
  .
  .
WORD(m) (1)

FUNCTION NAME (1) =====> NACK.6
CALLING PROCESS (2)
RETURN CODE (1)
```

WRITE expr [exprlst] <ONFAIURE stat>

Writes a number of conseccutive words into a data object. Expression 'expr' must evaluate into capability that points to the data object. Expression list 'exprlst' specifies offset within the object and the number of words to be written. In the case of unsuccessful execution of the primitive, the statement or sequense of statements following the keyword ONFAILURE is executed. If the ONFAILURE is not specified and the call fails, the calling process terminates.

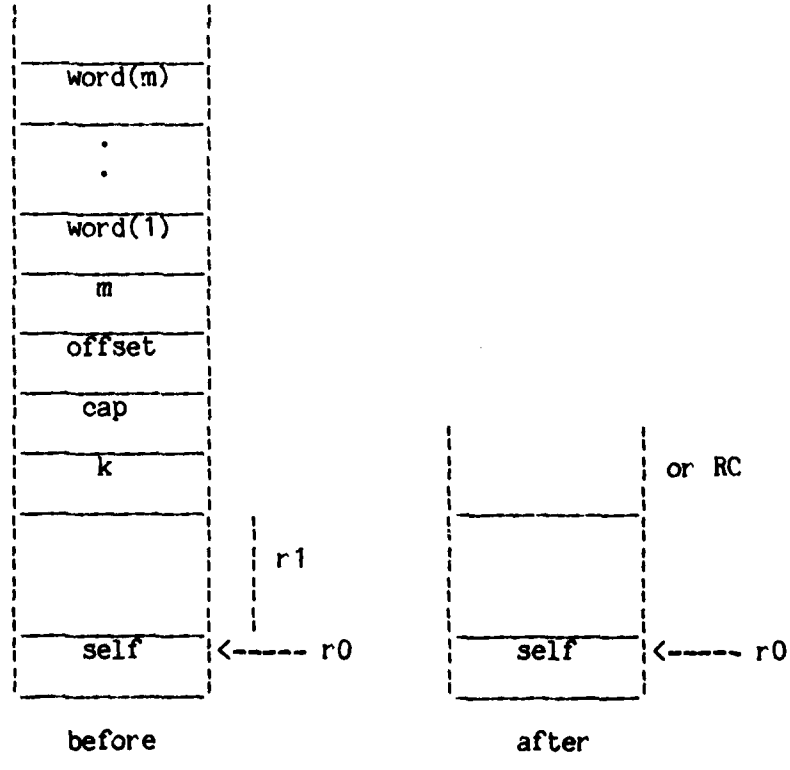


Figure 22: Data Segment for the process executing WRITE

k - number of arguments on the process's stack

cap - capability for the data object

m - number of words to be written

If, following the call, the value of the register r1 is -1, the call was successful and the calling process expects no arguments on its data segment.

If, following the call, the value of the register r1 is 0, the call was unsuccessful, and the calling process expects Return Code to be placed on its data segment.

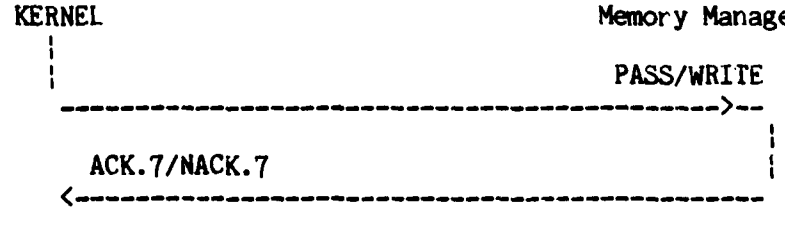
Return Codes:

RC = 1 nonexisting capability specified by the calling process

3 the capability specified by the calling process maps into invalid object name

5 segmentation exception - the caling process tries to write beoynd the limit specified by the limit field of the corresponding Segment Descriptor Register

Kernel - Memory Manager protocol:



Format of the message sent by the Kernel and received by Memory Manager's entry WRITE:

```
FUNCTION NAME (1) =====> WRITE
CALLING PROCESS (2)
OBJECT NAME (1)
OFFSET (1)
NUMBER OF WORDS (1)
WORD(1) (1)
  .
  .
WORD(m) (1)
```

Format of the message sent by the Memory Manager and received by Kernel's entry ACK.7 or NACK.7 :

```
FUNCTION NAME (1) =====> ACK.7
CALLING PROCESS (2)
```

```
FUNCTION NAME (1) =====> NACK.7
CALLING PROCESS (2)
RETURN CODE (1)
```